

# OPTIMISING JAVA PROGRAMS THROUGH BASIC BLOCK DYNAMIC COMPILATION

A MODIFIED VERSION OF A THESIS SUBMITTED TO  
THE UNIVERSITY OF MANCHESTER  
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY  
IN THE FACULTY OF SCIENCE AND ENGINEERING

September 2002

By  
Ian A. Rogers  
Department of Computer Science

# Contents

<b>Abstract</b>	<b>11</b>
<b>Declaration</b>	<b>12</b>
<b>Copyright</b>	<b>13</b>
<b>The Author</b>	<b>14</b>
<b>Acknowledgements</b>	<b>15</b>
<b>Alterations</b>	<b>16</b>
<b>1 Introduction</b>	<b>17</b>
1.1 The problem . . . . .	17
1.2 The solution . . . . .	18
1.3 Contributions . . . . .	21
1.4 Outline . . . . .	22
<b>2 The Java Virtual Machine</b>	<b>23</b>
2.1 What is Java? . . . . .	23
2.2 Objects . . . . .	25
2.3 Class file . . . . .	28
2.4 Class library and native methods . . . . .	30
2.5 Linking and loading . . . . .	31
2.6 Exceptions . . . . .	32
2.7 Interpreter JVMs . . . . .	34
2.8 Hardware JVMs . . . . .	37
2.9 Just-In-Time and dynamic compilation JVMs . . . . .	38
2.9.1 Method inlining . . . . .	39

2.10	HotSpot . . . . .	41
2.11	Summary . . . . .	43
<b>3</b>	<b>Dynamic Binary Translation</b>	<b>45</b>
3.1	Overview . . . . .	45
3.2	Above operating system dynamic binary translators . . . . .	49
3.2.1	FX!32 . . . . .	49
3.2.2	Dynamo . . . . .	51
3.2.3	UQDBT . . . . .	52
3.2.4	DAISY . . . . .	54
3.3	Between operating system dynamic binary translators . . . . .	54
3.3.1	VEST and mx . . . . .	55
3.3.2	Macintosh application environment . . . . .	55
3.3.3	Wabi . . . . .	56
3.3.4	VMWare . . . . .	56
3.4	Below operating system dynamic binary translators . . . . .	57
3.4.1	Transmeta . . . . .	57
3.5	Other directions . . . . .	58
3.6	Summary . . . . .	60
<b>4</b>	<b>Dynamite</b>	<b>61</b>
4.1	The Dynamite fuse module . . . . .	61
4.2	The Dynamite front-end module . . . . .	62
4.2.1	Building Dynamite's intermediate representation . . . . .	62
4.2.2	Substitute calls . . . . .	63
4.3	The Dynamite kernel module . . . . .	63
4.3.1	Control . . . . .	63
4.3.2	Basic block cache . . . . .	64
4.3.3	Basic block compatibility . . . . .	64
4.3.4	Group blocks . . . . .	65
4.3.5	Dead code elimination . . . . .	65
4.3.6	Constant propagation . . . . .	65
4.3.7	Value-specific optimisation . . . . .	65
4.3.8	Code duplication . . . . .	66
4.4	Back-end . . . . .	66
4.4.1	Instruction scheduling . . . . .	66

4.4.2	Idiom recognition . . . . .	68
4.4.3	Code generation . . . . .	68
4.5	Summary . . . . .	68
<b>5</b>	<b>Dynamite JVM</b>	<b>70</b>
5.1	Instruction decoding . . . . .	70
5.1.1	Expression stack . . . . .	71
5.1.2	Control-of-flow bytecodes . . . . .	73
5.2	Exceptions . . . . .	75
5.3	Object layout . . . . .	77
5.4	Class loader . . . . .	79
5.4.1	Boot strapping . . . . .	79
5.4.2	New . . . . .	79
5.4.3	Checkcast and instanceof . . . . .	80
5.5	JNI . . . . .	81
5.6	Threading . . . . .	81
5.7	Related work . . . . .	82
5.8	Summary . . . . .	84
<b>6</b>	<b>Inter-Procedure Optimisation</b>	<b>85</b>
6.1	The method inlining problem . . . . .	85
6.2	Sliding register-window scheme . . . . .	87
6.3	Fixed register-window scheme . . . . .	87
6.4	Choice of register-window scheme . . . . .	91
6.4.1	Introduction of benchmarks . . . . .	91
6.4.2	Number of fix-up blocks for fixed register-window scheme . . . . .	92
6.4.3	Number of retranslations for sliding register-window scheme . . . . .	96
6.4.4	Coverage of fixed register scheme . . . . .	99
6.4.5	Conclusion . . . . .	100
6.5	Summary . . . . .	101
<b>7</b>	<b>Recursion</b>	<b>104</b>
7.1	Lazy recursion detection . . . . .	104
7.1.1	Single recursion . . . . .	107
7.1.2	Direct mutual-recursion . . . . .	108
7.1.3	Indirect mutual-recursion . . . . .	110

7.2	Fixing the call stack . . . . .	111
7.2.1	Immediate stack fix-up technique . . . . .	111
7.2.2	Delayed stack fix-up technique . . . . .	112
7.2.3	Chosen scheme . . . . .	114
7.2.4	Example stack fix-up . . . . .	114
7.3	Performance . . . . .	115
7.3.1	Cost of recursion . . . . .	115
7.3.2	Cost of stack fix-up . . . . .	119
7.3.3	Performance comparison . . . . .	120
7.4	Summary . . . . .	125
<b>8</b>	<b>Overall System Performance</b>	<b>126</b>
8.1	Break down of JVM performance . . . . .	127
8.2	Translation saving . . . . .	129
8.3	Performance of translated code . . . . .	130
8.3.1	Externally measured performance . . . . .	130
8.3.2	Internally measured performance . . . . .	131
8.3.3	Analysis . . . . .	133
8.4	Translation performance . . . . .	135
8.5	Back-end performance . . . . .	139
8.6	Summary . . . . .	144
<b>9</b>	<b>Summary and Conclusions</b>	<b>145</b>
9.1	Summary . . . . .	145
9.1.1	Background . . . . .	145
9.1.2	Design . . . . .	145
9.1.3	Experiments . . . . .	146
9.2	Conclusions . . . . .	147
9.3	Future work . . . . .	148
9.3.1	Threading . . . . .	149
9.3.2	Garbage collection . . . . .	149
9.3.3	Exception handling . . . . .	150
9.3.4	Class library . . . . .	151
9.3.5	Memory usage . . . . .	152
9.3.6	Latency . . . . .	153
9.3.7	Performance . . . . .	154

9.4	Final remark . . . . .	154
<b>A</b>	<b>feJAVA Dynamite JVM Front-End</b>	<b>155</b>
<b>B</b>	<b>Result Data</b>	<b>157</b>
B.1	Number of fix-up blocks for fixed register-window scheme . . . . .	157
B.2	Number of retranslations for sliding register-window scheme . . . . .	157
B.3	Input to Takeuchi function . . . . .	160
	<b>Bibliography</b>	<b>162</b>

# List of Tables

3.1	Translation environments . . . . .	48
6.1	Benchmark translation and execution bytecode counts . . . . .	93
6.2	Percentage of method call bytecodes translated and executed in fixed register-window scheme . . . . .	95
6.3	Fixed register-window copying overhead per virtual call . . . . .	96
6.4	Overall sliding register-window statistics . . . . .	96
7.1	Descriptor abbreviation descriptions . . . . .	115
7.2	Recursive instruction counts . . . . .	117
7.3	Recursive instructions as a percentage of their base instruction . .	117
7.4	Recursive bytecodes as a percentage of the total instruction mix .	117
7.5	The total size of frames saved and loaded from the memory stack by recursive invoke bytecodes . . . . .	118
7.6	The average size of frames saved to the memory stack by recursive invoke bytecodes . . . . .	118
7.7	Stack fix-up cost . . . . .	119
8.1	Number of translated bytes of a JIT compiler and the Dynamite JVM . . . . .	129
8.2	Reduction in translated bytes by basic block compilation . . . . .	129
8.3	Summary of execution performance . . . . .	134
8.4	Translation cost: total execution time . . . . .	137
8.5	Translation cost: inner loop execution time . . . . .	137
8.6	Translation cost: increase in total execution time per loop unroll .	138
8.7	Translation cost: increase in inner loop execution time per loop unroll . . . . .	138

B.1	Number of method call bytecodes translated and executed in fixed register-window scheme . . . . .	158
B.2	Number of translations necessary for methods using the sliding register-window scheme . . . . .	159
B.3	Number of translations necessary for methods using the sliding register-window scheme (_202_jess) . . . . .	160
B.4	Takeuchi input . . . . .	161

# List of Figures

1.1	Java source code of a small test program . . . . .	19
2.1	Relationship between the Java programming language and the Java environment . . . . .	24
2.2	Example bootstrapping problem . . . . .	27
2.3	Class file structure . . . . .	29
2.4	Example Java code handling exceptions . . . . .	32
2.5	Bytecode example . . . . .	35
2.6	Comparison of bytecode and RISC . . . . .	37
2.7	Example of method inlining . . . . .	40
3.1	Dynamic binary translation environments . . . . .	49
3.2	The FX!32 System . . . . .	50
3.3	Dynamo model of execution . . . . .	52
3.4	VEST and mx execution model . . . . .	55
4.1	Dynamite modules . . . . .	62
4.2	Interaction of Dynamite modules . . . . .	63
4.3	Instruction scheduling on an Intel Pentium processor . . . . .	67
4.4	Estimated performance of code vs. translation time . . . . .	69
5.1	Example Dynamite JVM translation from bytecode to IR . . . . .	72
5.2	Expression stack during translation . . . . .	72
5.3	IR for an array out-of-bounds exception . . . . .	76
5.4	General layout of a Dynamite JVM object . . . . .	77
5.5	Bytecodes for a synchronised method . . . . .	82
5.6	Java array accesses compared to those of Fortran . . . . .	84
6.1	Method inlining . . . . .	86

6.2	Sliding register-window example . . . . .	88
6.3	Fixed register-window example . . . . .	90
6.4	Fix-up overhead for fixed register-window scheme . . . . .	94
6.5	Number of translations required for the sliding register-window scheme . . . . .	98
6.6	Approximate number of registers capturing a percentage of a total program execution . . . . .	103
7.1	Dynamic loading altering recursive behaviour of code . . . . .	106
7.2	Reducing locals saved to stack . . . . .	108
7.3	An example showing the need of recursion fix-up . . . . .	109
7.4	Complex recursive example . . . . .	110
7.5	Delayed stack fix-up . . . . .	112
7.6	Pathological case for immediate stack fix-up . . . . .	114
7.7	Example of stack fix-up . . . . .	116
7.8	Takeuchi benchmark program . . . . .	120
7.9	Takeuchi method bytecode basic block graph . . . . .	121
7.10	Takeuchi function translated into IR . . . . .	122
7.11	Takeuchi execution results . . . . .	123
7.12	Takeuchi execution results (more bytecodes) . . . . .	124
8.1	Execution cost: total execution time for values of $E$ up to 50,000 .	131
8.2	Execution cost: total execution time for values of $E$ up to 100 . .	132
8.3	Execution cost: total execution time for values of $E$ up to 2,800,000	132
8.4	Internally measured execution cost . . . . .	133
8.5	Translation cost: total execution time . . . . .	137
8.6	Translation cost: inner loop execution time . . . . .	138
8.7	Dynamite JVM translation of hashCode method . . . . .	140
8.8	String hashCode method target code . . . . .	141
8.9	Optimised string hashCode method target code . . . . .	143
B.1	Function to calculate the number of bytecodes executed by the Takeuchi function . . . . .	161

# Abstract

This thesis presents new techniques designed to speed up the execution of Java programs within a Java Virtual Machine (JVM). It presents a JVM based on Dynamic Binary Translator (DBT) technology with developments to minimise unnecessary translation and produce optimised hot paths.

Existing JVM runtime compilation technology compiles classes, methods or portions of a method. This can make the compilation process inefficient as not all the compiled code will be executed. It does mean however, that mapping from Java to a target machine's code can be performed simply at a small number of boundaries.

The thesis develops low overhead mechanisms for executing Java bytecode by compiling it a basic block at a time, thus avoiding unnecessary translation. It also presents mechanisms for grouping translated blocks into larger blocks based on execution statistics. By a novel register-mapping technique inter-procedural optimisation at a low overhead is achieved. As all optimisations are directed by execution statistics, near optimal ordering of code and register usage can be produced quickly.

The thesis shows that basic block translation affords a saving of up to 18% fewer bytcodes being translated than if a method were translated. It is also shown that this is achieved with little overhead in the translated code. A set of representative single threaded Java benchmarks are used for analysis. The overall system performance is compared to the HotSpot client and server JVMs. The measured execution time is from 3.163 to 19 times slower. The thesis discusses how, with improved translation performance, this can be significantly increased. Combined with a more optimal set of library methods and translation savings, performance can be comparable and potentially better than state-of-the-art JVMs.

# Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

# Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this modified thesis is vested in Transitive Limited, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of Transitive Limited, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the CFO, Transitive Limited.

# The Author

Ian Rogers received a Bachelor of Science honours degree in Computer Engineering in 1998 from the University of Manchester. At the same time he was awarded the Edward's prize for best Computer Engineering student.

In June 2000 he received a Master of Philosophy degree for research carried out in the department of Computer Science at the University of Manchester.

Ian has contributed to the Dynamite dynamic binary translator project since 1997. His first project was a front-end for the LARD virtual machine, an in-house tool for asynchronous hardware development. Ian's Master's project was the initial development and background work to the Dynamite JVM.

Ian's research for his Masters and Ph.D. has been funded by the Engineering and Physical Sciences Research Council. Ian was also in receipt of the University of Manchester Atlas scholarship.

# Acknowledgements

Thank you to my supervisor and advisor Professor Ian Watson.

Thanks to my friends who helped with me to write this thesis, Ahmed El-Mahdy, Greg Wright, John Rooksby, Andrew Bardsley, Richard Stokes, their time, feedback and help is always invaluable.

Thanks to my friends in the Transitive Technologies team who, along with myself, continue to make Dynamite a viable commercial and research tool. I'd particularly like to thank Daniel Owen, John Sandham, Miles Howson, Dave Gilbert, Paul Knowles, Ian Cottam, Gavin Baraclough, Nick Jefferson, Ian Bolton, Jeff Kass, Neil Hegarty, David Ung, Frank Weigel, James Walker, John Fitzgerald, Assad Hashmi, Georges Guy, John Turner and Geraint North for their help with my work.

Thanks to the doctors and nurses of the Rheumatology department at the Manchester Royal Infirmary.

Thanks to all my other friends, my family and those who have helped me at the oak-tree house.

# Alterations

This thesis has been altered from that presented for examination in September 2002. Alterations were made with consultation with the Ph.D. examiners in February 2003 and Transitive Limited in March 2006.

This work presents information about a prototype implementation of a Java Virtual Machine called the Dynamite JVM. The implementation of the Dynamite JVM cannot be used to infer the implementation of any Transitive Limited product. The performance results are those purely of the Dynamite JVM and cannot be used to infer the performance of any Transitive Limited product.

# Chapter 1

## Introduction

### 1.1 The problem

Java is a popular programming language and program environment. As described in chapter 2, Java programs are executed by a Java Virtual Machine (JVM)<sup>1</sup> that provides the Java environment. Java programs are distributed in a platform independent class file. The Java bytecodes contained in a class file can be executed in three ways.

- Interpretation: Each Java bytecode is interpreted in software and its effect on the JVM calculated.
- Hardware: Each Java bytecode is executed by a dedicated piece of hardware which represents the JVM.
- Dynamic compilation: Java bytecodes are compiled onto a target computer architecture. The compiled code is then executed instead of the original Java bytecode.

Interpreters are easily implemented in software and have been used to provide the reference JVM. Interpreting a Java bytecode takes at least several CPU clock cycles; this means it is unsuited to executing Java programs that require high performance.

---

<sup>1</sup>Static translation of Java programs to a target computer architecture is possible, however, this breaks the Java model and programs translated in this way do not execute in a true Java environment.

Executing Java bytecodes in hardware is typically faster than interpreting Java bytecodes. Many bytecodes are complex and may need to be interpreted by an additional controlling processor. In particular, method calls to the underlying system cannot be dealt with by Java bytecode alone.

Dynamic compilers are executed in parallel to the bytecode they have translated. They are required to produce high quality code quickly. The majority of existing dynamic compilers work by translating a method the first time it is called. This slows the Java environment down initially but, hopefully, in the long run will prove to have made the program execute faster. State-of-the-art JVMs [Sun99b] are using multiple levels of compilation; they aim to translate quickly at first and then target expensive compiler optimisations.

This thesis considers a dynamic compiling JVM. The problems this JVM must address are to execute Java programs faster, to have little latency in interactive applications<sup>2</sup>, to use little memory and to support all of the rich set of features of the Java programming language and Java environment.

## 1.2 The solution

The main thesis presented is:

Method based translation of Java bytecodes unnecessarily translates bytecodes that will never be executed, wasting both time and space. By translating at the basic block level, optimisation can be focused away from bytecodes that will never or will be infrequently executed. Execution statistics of the basic block allow for profile directed optimisation to schedule basic blocks and create optimal code layout for hot paths.

Although basic block translation of instructions happens in existing dynamic binary translation environments, dynamic Java compilers are method oriented<sup>3</sup>. Method compilation reflects conventional compilation, with the addition of execution information. Basic block compilation is different in that it focuses on a smaller unit of execution and groups these to form larger blocks that are then

---

<sup>2</sup> Latency is the time between requesting an activity and that activity being performed (see section 8.1).

<sup>3</sup>Some JIT compilers, such as LaTTe [LYK<sup>+</sup>99], do not translate exception handling regions of code as they assume they are infrequently executed.

optimised. There is a perceived cost in creating this information about larger blocks, and although trace scheduling is a natural consequence of a basic block compilation model, all existing JVMs are method oriented. This work shows the benefits of basic block translation, and that it can be achieved with little overhead.

```

class test
{
    public static void main (String args[])
    {
        /* Basic block 1 */
        int i, a[];
        a = new int[100];
        i=0;
        /* Basic block 2 */
        while(i<100){
            /* Basic block 3 */
            a[i]=i*i;
            i++;
        }
        /* Basic block 4 */
    }
}

```

Figure 1.1: Java source code of a small test program

To understand this approach at a high-level, some sample Java code is shown in figure 1.1. The Java source code is a purely illustrative example of a Java program comprising a method of a few basic blocks. The program does no more than create an array of integers and initialise each integer to the square of its index. As the first stage of compilation, a typical Java compiler will compile the program into 4 basic blocks (the comments in the source code show the basic blocks, the numbering is arbitrary). The first basic block performs the array creation and loop initialisation, then it performs an unconditional branch to the second basic block. The second basic block is the loop test, depending on the result of the test the third or the fourth basic block may be executed next. The third basic block performs the loop body and unconditionally branches to the second basic block. The fourth basic block exits the method returning back to either the JVM system or the calling method.

The next stage for a Java program is to be executed in a JVM. We consider the execution of the program within a JVM that uses a dynamic compiler. After the JVM has created the environment used to execute the program within, it calls the method called `main`. The method call tells a Just-In-Time (JIT) compilation [Wil02] system to compile the method as the method has not been compiled previously. The exact process of compilation varies between JVMs. A template compiler would swap each Java bytecode for equivalent target processor instructions. A second pass would be necessary in order for forward branches to be resolved. More sophisticated compilation systems remove the overhead of the Java stack [Kra98]. A state-of-the-art compilation system, such as HotSpot [Sun99b], would initially interpret the bytecode to see if a significant proportion of program time is being spent in this method. At a trigger point, HotSpot will compile the method in parallel to the interpretation. When compilation is complete, if the same bytecodes are executed, the compiled code will be used instead of using the interpreter. HotSpot is described more fully in section 2.10.

The example in figure 1.1 executes all its basic blocks. Blocks two and three are good candidates for compilation as they are executed 100 times. However, blocks one and four are poor candidates for compilation as they are executed once. Java's exception mechanism means there are often several basic blocks in a method that will only ever get translated in an exceptional circumstance. A method based compiler has to compile an entire method. Optimisations within a method based compiler can be directed by execution statistics, but unnecessary basic block compilation is unavoidable.

Object-oriented programmers are advised to keep methods short as good programming practice [Sha97]. Studies of object-oriented software such as [BKP99] and of the Java class file [AP98] have shown that basic block sizes tend to be shorter. This means the scope for optimisation within a method is small. This is solved by using expensive optimisations such as method inlining, where the body of the called method is placed at its call site (thus removing the method boundary). This can lead to code bloat (the code for the method is repeated several times), and this has implications on the performance of the CPU's instruction cache. Due to the nature of the Java environment, method inlining cannot be performed prior to execution in the JVM or else the JVM would not properly support dynamic loading and linking of bytecode.

In our system basic blocks are translated as and when needed, as opposed to methods, thus avoiding unnecessary compilation. Further, method inlining is performed by grouping a call site basic block with its called basic block and all other frequently executed basic blocks in those methods. Parameter passing is minimised in a two-pass compilation system, and by using heuristics in the kernel of our system we avoid unnecessary code bloat from inlining. As basic block layout is not driven by method layout, but execution profile heuristics, we can trace-schedule instructions and maximise performance by optimising, for example, for the CPUs instruction cache.

### 1.3 Contributions

The main contributions in this dissertation are:

- We investigate the design space of a basic block based dynamic compilation system for JVMs by designing and implementing one. By compiling at the basic block level, we reduce the number of bytecodes that need to be translated compared with conventional Java JIT and dynamic compilers. This JVM has full support for exceptions and native methods, with threading at the design stage. In-stack replacement is avoided by performing only dynamic compilation, but performing it with differing amounts of optimisation.
- We design and implement a register allocation algorithm that allows the visibility of multiple method's registers to a profile based optimisation mechanism. This enables method inlining to be performed and, at the same time, the optimisation algorithm is 'aware' it is dealing with the same basic block (not a duplicate as in traditional inlining/specialisation). This means the optimisation algorithm can choose to specialise to the previous basic block (the call site) or it can re-use the basic block to avoid code bloat.
- We design and implement a fast, lazy recursion detection algorithm so that the JVM system does not bloat our register pool, whilst not requiring any forward parsing of Java class files to avoid this; this was a vital development for our register allocation algorithm.

## 1.4 Outline

The first chapters of this thesis describe the background to the research. Chapter 2 introduces the JVM and describes key technologies for its usability and speed. Chapter 3 describes dynamic binary translation technology. Chapter 4 describes Dynamite, a research and commercial dynamic binary translator, which is used as the dynamic compiler for this research.

The next three chapters describe the Dynamite JVM and investigate interprocedural optimisations. Chapter 5 describes the Dynamite JVM created as part of this research. Chapter 6 introduces the register algorithm used by the Dynamite JVM; chapter 7 describes how this optimisation works with recursive methods. These chapters measure the implemented techniques and demonstrate their low cost.

The final two chapters of this thesis look at the performance of the Dynamite JVM and the further work necessary to improve it. Chapter 8 investigates the performance of the Dynamite JVM compared to the HotSpot client and server JVMs. Chapter 9 describes further research into the Dynamite JVM and the Dynamite dynamic binary translator. In conclusion chapter 9 describes how the research presented and future work will contribute to research into JVMs and dynamic binary translators.

## Chapter 2

# The Java Virtual Machine

This chapter describes Java and the components of it. It is oriented toward the implementation of a JVM. Section 2.1 focuses on what Java is and serves as an introduction to the specific Java internals issues discussed in sections 2.2, 2.3, 2.4 and 2.5. After the internals of Java are described the focus is shifted toward the implementation of a JVM in sections 2.7, 2.8 and 2.9. Method inlining, a key compiler optimisation, is described in section 2.9.1. Finally, a complete state-of-the-art JVM is discussed in section 2.10.

### 2.1 What is Java?

The name Java has, unfortunately, been used in an ad hoc manner. A person browsing the web understands Java to be a kind of program they download and run in their web browser. A computer science student understands Java to be a programming language and set of standardised libraries. There are essentially two views of Java, either as a programming language or as an execution environment. The interaction between the two is shown in figure 2.1.

Sun defined Java in their Java language white paper [GM95] to be:

a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded and dynamic language.

A design objective of the Java programming language was to keep the language simple. This decision was made in the context that at that time most programs were written in the C++ programming language [Str86]. As Java only provides a

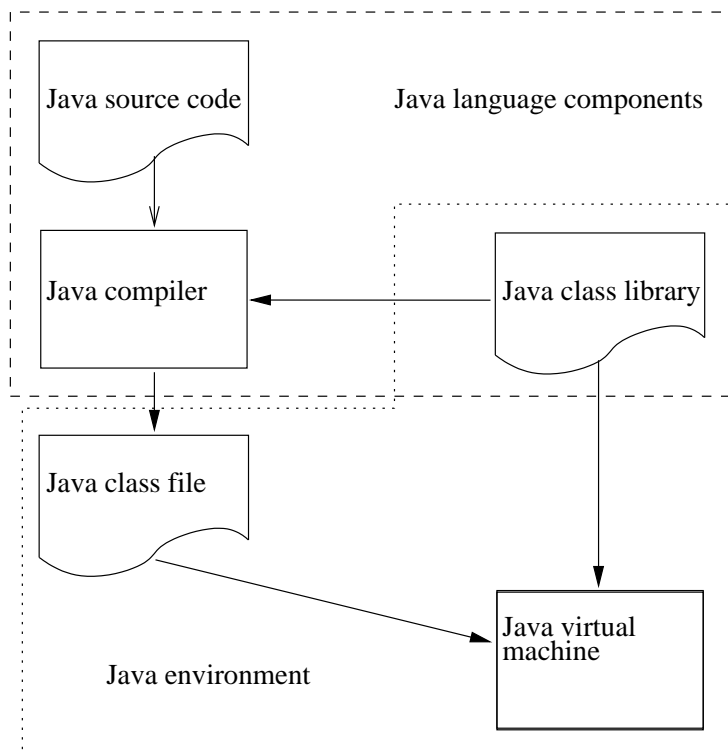


Figure 2.1: Relationship between the Java programming language and the Java environment

subset of the programming language constructs that are available in C++, it can be regarded as simpler in that sense. This doesn't mean that Java programs are simpler to execute or that Java is necessarily easier to program. This is apparent, for example, by the large size of the set of standard libraries provided for Java.

Section 2.2 describes the object-oriented nature of the Java language and JVM further.

The distributed features of Java enable programs to be downloaded over the Internet. They also provide Java with a rich set of network library functions. The distributed nature of Java is discussed further in section 2.5.

Interpretation is a feature of how a Java program is run, that gives it portability and architectural neutrality. The instructions (or bytecodes as they are known) contained in the compiled class file are interpreted by a virtual machine rather than run on the machine natively. The fact that bytecodes are interpreted enables the compiled code to be portable between different virtual machines built for different computer operating systems and instruction set architectures. The

only code that needs to be ported is the virtual machine. Interpretation is discussed further in section 2.7.

Robustness was an objective of the Java programming language and is supported by the use of exceptions. Exceptions are described in section 2.6.

The downside to interpreting Java programs is that the programs will execute slowly. The ability to achieve high-performance is discussed further in section 2.7.

Multithreading allows parts of a program (called threads) to execute at the same time. By having threads designed into the Java programming language, programmers can write multithreaded programs without having to rely on thread-support libraries. This allows programmers to avoid long-winded syntax and maintenance of thread data structures. By having threads as part of the Java environment portability concerns are dealt with by the JVM.

Java is a dynamic language as it allows code to be dynamically loaded and linked into the executing program at runtime. This is described further in section 2.5.

## 2.2 Objects

Java programs are object-oriented. An object is viewed in object-oriented design as a black box that contains state which has defined behaviours for receiving and sending messages. The message passing mechanism is typically a method call, however, in Java information can also be passed by reading and writing to publicly accessible variables (state).

Objects are instances of data-types called classes. Classes are able to inherit attributes from other classes, this is done in the Java programming language using the **extends** keyword. All classes are subclasses of the base class called `java.lang.Object`. A new class definition can redefine or, as it is commonly referred to, override methods from a superclass. A method that takes an object of a particular class as an argument can have a subclass of it passed in also. If a method is to be called on an object then the exact method called depends on the class type of the object. In Cardelli and Wegner's model, this is called inclusion polymorphism [CW85]. Java also supports parametric polymorphism (more commonly referred to as overloading) whereby a method is uniquely identified by not only its name but also its parameter types. The JVM calls a parameter list a method descriptor.

Every time an object is created an initialiser is run which optionally takes parameters. Java initialises primitive types (see later) to default values, an object initialiser goes on to overwrite these values.

Multiple inheritance in Java is a compromise. The Java programming language only allows a class to have one superclass unlike other object-oriented programming languages, such as C++, that allow a class to have multiple superclasses. Having multiple superclasses complicates method calls (message dispatch). Programming languages that compile to a static binary file can calculate what methods can be called and optimise the dispatch mechanism appropriately. As Java is a dynamic language (see section 2.5) it was decided to keep the message dispatch mechanism quick and simple by only allowing single inheritance.

The compromise in Java is that it supports interfaces. An interface is a class that defines methods but provides no implementation, they are implicitly **abstract** and all methods are both **public** and **abstract**. An interface can have no variables but it can define constants. As an interface provides no implementation or variables it cannot be instantiated. As with class inheritance, interfaces do provide a way of describing an abstract feature that several classes can **implement**. A class is permitted to implement as many interfaces as it requires. As with classes, interfaces can inherit features from another interface. As interfaces do not require any state they do not alter the object layout for a class. They only complicate the execution of a Java program by requiring a method dispatch mechanism that first has to select the appropriate interface for the method call and then the appropriate method.

As well as methods and state which are associated with objects, Java allows methods and state to be associated with a class. The keyword **static** is used to declare these in the programming language. The method **main** is a static method that is first called by the JVM. A typical main method creates the programs objects and then passes control to them.

In the Java environment there are certain global objects for performing primitive tasks such as string handling and file IO. These global objects are declared as static. They are initialised by a set of rules defining when a class is to be loaded. A special method called a class initialiser is called the first time a class is loaded, this method sets up these global variables. The phase of execution where the JVM is setting up global variables is called the bootstrapping phase. Cyclic dependency problems can be encountered in the bootstrapping phase as shown

in figure 2.2.

```

/* The Java String library is one of the first libraries loaded.
It
uses a Hashtable as an internal lookup table for strings as the Java
specification states there should only exist one copy of a string
with a particular value. */

class String {
    private static final Hashtable intern_table [];
    ...
}

/* This example Hashtable creates a bootstrapping problem as it
declares a String that the JVM needs to intern. There exists a cyclic
dependency where the Hashtable relies on the String to be initialised
and the String relies on the Hashtable to be initialised. */

class Hashtable {
    private static final String copyright =
        "The University Of Manchester (C)";
    ...
}

```

Figure 2.2: Example bootstrapping problem

Primitive types such as integer have dual modes in Java. Initially they are not objects, however, an Integer object can be created. Primitive types also have primitive operations that can be performed on them, such as addition. This removes the message passing overhead from primitive types and means that aggressive optimisation of primitive types is unnecessary. This is unlike other object-oriented languages such as Self [US87]. Java supports boolean (1bit value), byte (8bit value), char (16bit value), integer (32bit), long (64bit integer), float (32bit floating point) and double (64bit floating point) primitive types. Boolean, byte and char primitive types are represented as integers on the Java stack and in local variables. The size of the boolean, byte and char primitive types only affects the implementation of memory operations on objects.

All data structures are represented by objects which are composed of either more objects or primitive types. Objects are always accessed by an object reference which is a pointer to the object data structure within the heap. Object references are a primitive type.

The size of an object in Java is determined by the primitive types it is composed of, the alignment rules of the processor and how the primitive types are arranged within the object. An object representation of booleans could use byte values as this can speed accesses to individual boolean values at the expense of compactness. Similar issues occur on computer architectures that don't allow unaligned memory accesses. An object must also hold a header which holds a pointer to class information for method dispatch and for supporting bytecodes that query the type of an object. Thread lock information is also held in the object header.

All object representations must enable method dispatch. As described earlier method dispatch must support interfaces and virtual (single inheritance) method dispatch. Optimisations of various components of the Java environment may affect the number of memory indirections to locate the method to execute, slowing the method dispatch mechanism.

## 2.3 Class file

Java programs are compiled into class files. The contents of a class file are illustrated in figure 2.3.

After the Java class files header, the constant pool contains all the values used to describe object and method layout. The fields section contains type information about fields within a class. The access flags enable the JVM to stop unauthorised accesses to an object. The access flags also declare whether a field is static (one per class) or an instance field (one per object). Similarly, the methods section describes the types of methods, their bytecode and their access permissions. The attributes section enables extra information to be passed within a class file. An example use for this section would be line number information for a debugger.

The class file format is very rich in the sense that it contains almost as much information as the original Java source code. It has been shown that it can be transformed back to Java code [PW97, AP98]. At the same time, bytecode within a class file has had few significant optimisations performed on it. The class file structure has to contain a lot of information as Java code compiled later has to be linked against it. This prevents standard compiler and object-oriented optimisations from being performed until the class files are within the JVM. Later

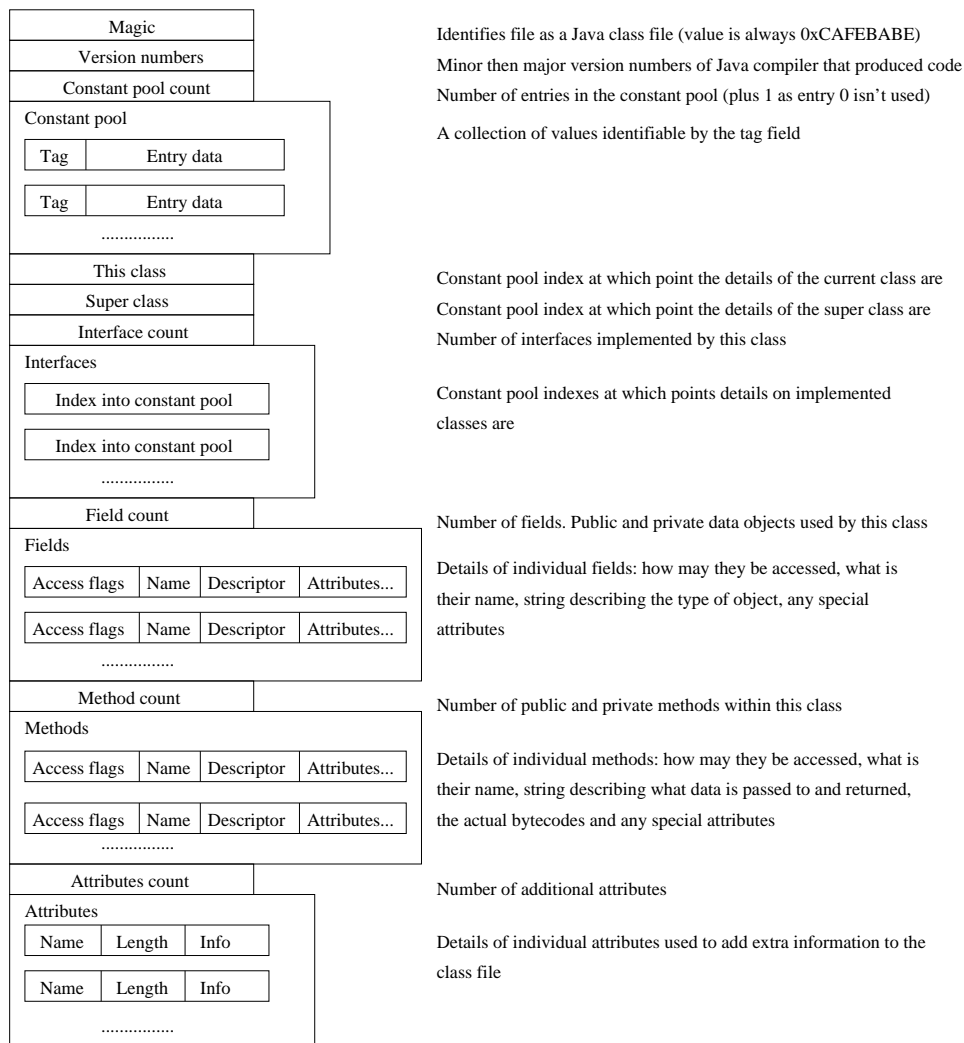


Figure 2.3: Class file structure

in the chapter these optimisations are discussed.

## 2.4 Class library and native methods

Java bytecodes have no mechanism for performing any input from or output to the operating system, instead certain methods within the class file are flagged as native and are not provided with implementations. When a native method is called it signals for the JVM to perform some library code that is not within the class file. The exact way the native method is called depends on the class library the JVM is using.

Two methods exist for calling native methods, the first is to have a high degree of integration between the JVM, its compiler and the native methods. This technique is used by various JVMs, two are described below.

The Kaffe JVM [Wil02] has its own class libraries that use a native method interface called the Kaffe Native Interface (KNI). Kaffe uses the same object representation as the C++ compiler used to compile the native methods. This poses numerous problems due to differences between the two languages and the need for the JVM to know the layout of objects within the C++ compiler.

Like KNI, the GCJ project [Fou02b] uses a native interface called Cygnus Native Interface (CNI) [Fou02a]. GCJ does not provide a true Java environment as it compiles Java statically to native code and therefore doesn't support features such as dynamic loading. The GCJ is part of the GNU Compiler Collection [Fou02c] which has a C++ compiler. CNI provides a mapping from Java to C++ so that native methods can be written in C++. The native methods must be compiled with a C++ compiler compatible with GCC. This places similar restrictions on the programmer that KNI does.

This is a high-performance technique, due to the large amount of integration, but is not very portable. Users are tied into using a particular compiler and only writing native code in a particular language.

The second method uses an application binary interface (ABI) that gives the native code hooks back into the JVM. Sun's Java Native Interface (JNI) [Sun96] provides such a mechanism, it passes around a Java environment that has the necessary hooks into the JVM. In this approach, to access a method or a variable, a function in the environment must be called. This technique is more portable than the first solution, which was language and compiler specific, the native

methods just need the ability to use the Java environment and call external methods. However, this method is less efficient as the extra function calls slow the native methods down.

To speed up using an ABI a JVM may implement some frequently called native methods as if they were bytecodes; compiling or interpreting them as a single unit without the method call overhead.

## 2.5 Linking and loading

A key part of a JVM is the class loader [LB98]. The class loader is responsible for loading the class files and integrating them into the JVM. The primordial class loader is the class loader provided by the JVM, however, Java has an extensibility feature that allows it to be replaced by class loaders with extra facilities. For example, the primordial class loader may only be able to load uncompressed class files and not be able to load class files from over the Internet. A library can replace the primordial class loader with a class loader of its own with these extra features incorporated. As part of the security features of Java, this separate class loader is given its own name space so none of its classes can conflict with system classes. By writing the class loader in Java and having the JVM load it, the class loader can take advantage of Java's rich library code.

Class resolution is the process of loading a class when it is required by a JVM. The JVM specification requires classes to be resolved in a lazy manner so that a class is only resolved the first time it is referenced. References are made to other classes in the information contained in the class file about what classes this class file extends and implements. To enable a class to be resolved, these other classes must be loaded and resolved. Other classes are also referenced in bytecodes, these should not be resolved until the bytecode is executed for the first time. This places an unfortunate requirement on the JVM's interpreter or dynamic compiler; they cannot resolve and optimise certain bytecodes ahead of time.

## 2.6 Exceptions

As introduced earlier, Java aims to provide a robust environment in which to execute code. The key feature of this is an exception mechanism that allows executing code to signal to the JVM that something has gone wrong. This happens in a precise manner [GJS00], meaning all statements prior to the exception will have completed at the point the exception happens. The JVM also detects when something internally has gone wrong by generating its own exceptions when, for example, an array boundary is over run, a NULL pointer is dereferenced or there is a problem loading a class. Exceptions in Java are modelled using an object that is a subclass of `java.lang.Throwable`. A Java program must declare which exceptions the code can throw unless it is a common form of exception. The common form of exceptions in Java are held in the `java.lang.Error` and `java.lang.RuntimeException` class hierarchy subtrees.

```
import java.lang.*;

class test
{
    private int bar (int [] a, int x)
    {
        return a[x];
        // Potential exception if x is beyond the array boundary
    }
    public void foo ()
    {
        int a[10];
        int z;

        try {
            z = bar(a,12);
        }
        catch (ArrayIndexOutOfBoundsException e) {
        }
    }
}
```

Figure 2.4: Example Java code handling exceptions

Figure 2.4 shows some Java code that will cause the JVM to generate an internal exception in the method `bar`. The JVM creates a `java.lang.ArrayIndexOutOfBoundsException` object. As with all exception

objects the JVM must create a stack trace to place in the object to support methods such as `java.lang.Throwable.printStackTrace()` [Sun02a]. Once an exception object is created the exception is thrown. The process of throwing an exception involves starting at the head of the stack trace and checking whether the exception was thrown in a try block. If the exception is in a try block then the JVM checks if there is a corresponding catch block for this class of exception or a catch block for an exception that is a superclass of this exception. In the example, the top method does not contain a try or a catch block so the JVM proceeds to the next method in the stack trace. The exception is re-thrown in this method at the point that the method that caused the exception was called. This continues until the stack trace is exhausted and then the JVM handles the exception, typically printing the name of the exception and the stack trace.

The approach of handling exceptions above is described in the JVM specification [Sun95] and is described as the stack unwinding technique. To improve exception performance stack cutting can be used. Stack cutting places an exception and the handler's entry point onto an internal stack whenever a try region is entered. This stack entry is popped from the stack when the try region is left. Instead of re-throwing exceptions this approach can more quickly yield the correct handler. However, pushing and popping onto the exception stack happens in the normal path of program execution and adds to the execution time of a method.

A suggested approach [OKN01] that tries to incorporate the best parts of each exception handling system is to use a feedback directed optimisation that places exception throwing code for methods that frequently throw exceptions at the place the exceptions are thrown. This approach adds to the complexity of the JVM interpreter or dynamic compiler (as described in the following sections) as they must perform an analysis and modify the interpreted or executed code. If the extra time spent is not regained by a more efficient exception handling system then the JVM will have slowed down.

Analysis of exceptions allows for instructions to be rescheduled around exceptions and the amount of state generated when an exception is thrown to be reduced in certain circumstances [GCH00]. Exception handler prediction creates a fast path through to a predicted exception handler on the basis that the handler for a thrown exception will be the same for each exception as long as the handler is in the same method [LYK<sup>+</sup>00]. This optimisation is greatly improved by method inlining (see section 2.9.1) but is still limited as to the kind of exceptions

it can optimise.

## 2.7 Interpreter JVMs

Java's portability as a language stems from the fact it is compiled into a machine independent bytecode format that is intended to be interpreted within a JVM. To speed interpretation the operation represented by each bytecode is encoded into a single byte at the beginning of the instruction. Following the operation are zero or more operand bytes describing the operation. Common operations and operands are encoded into one bytecode to reduce the size of the compiled code.

A key feature to the bytecodes machine independent format is that it uses a stack to carry out operations instead of registers. By using a stack, processors with few addressable registers, such as the Intel IA32 instruction set architecture [Int87], are able to use their indirect addressing modes to emulate the stack. Instruction set architectures with more registers, such as the SPARC architecture [SPA92], can map the stack into registers. As the bytecodes use the top of the stack implicitly, bytecodes do not have to encode which register or memory address to use for an operation. This helps to improve the bytecodes code density. Mapping the stack to registers is discussed further in section 2.9.

As well as a stack, Java bytecodes can access a pool of local variables and swap them in and out of the stack to have general purpose computations performed on them. One exception to this is the `iinc` bytecode that directly increments a specified local variable.

Bytecodes are addressed by a virtual machine's program counter (PC) register. All PC values within a method are relative to the beginning of the method. The PC is used to calculate branch targets but its value can never be read. This allows a target machine to replace the PC with a value to speed up hardware, translation or interpretation.

An example of some Java bytecodes is shown in figure 2.5 generated by the `javap` tool [Sun02b].

The downside to interpreting Java programs is that programs will execute slowly. A Java interpreter has to fetch a bytecode, decode it (branch to a location that has the code to implement this particular bytecode) and then execute the code to perform the bytecode. A hardware JVM would fetch the bytecode

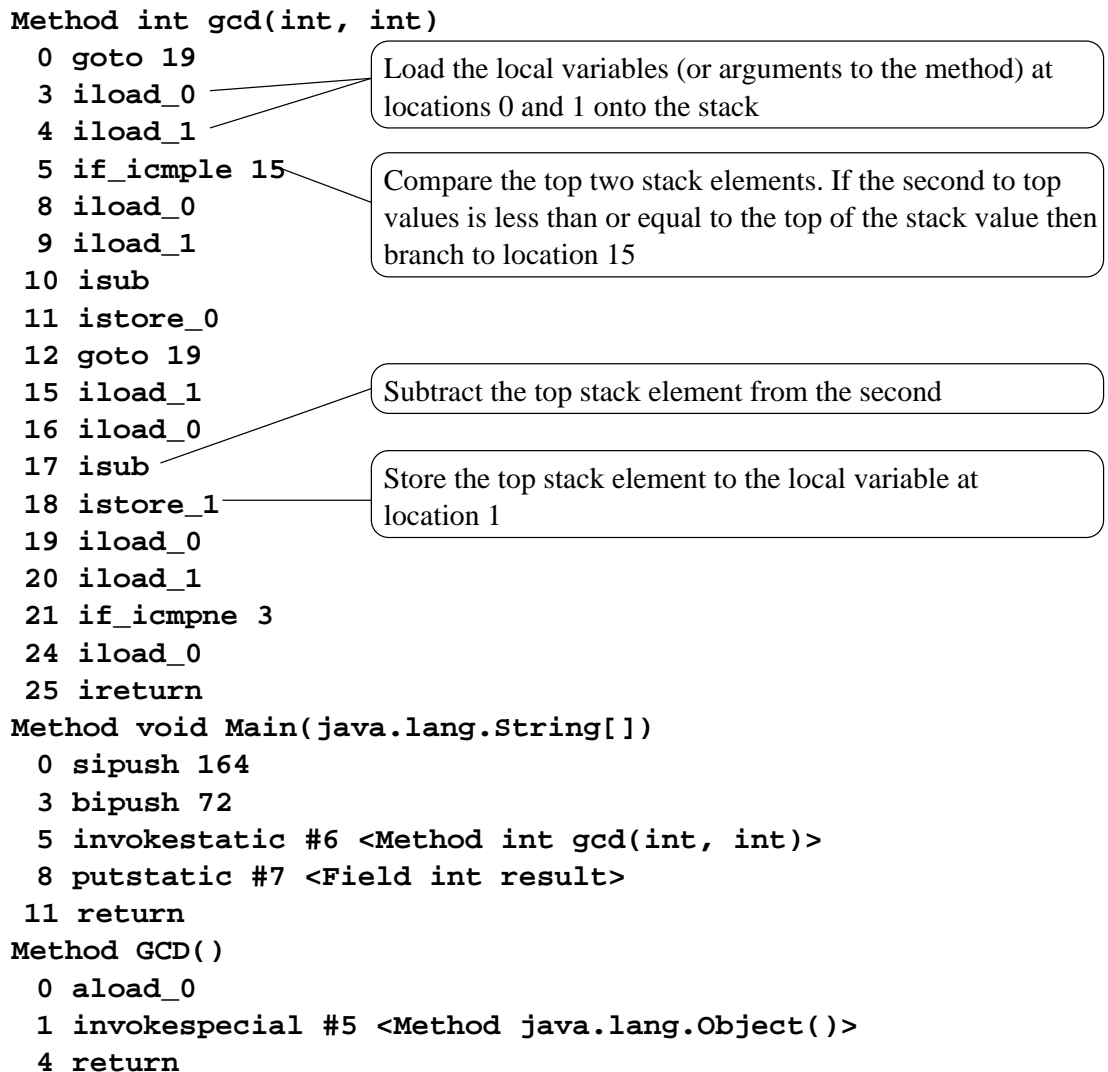


Figure 2.5: Bytecode example

and then execute it. This is analogous to executing a single instruction. An interpreter must perform several instructions to perform the load, fetch and execute operations.

High-performance was a stated goal of Java. Often application performance is not dependent on the speed of the actual program but more on the speed of surrounding libraries and the operating system. Java's class libraries are therefore designed with high-performance in mind. Interpretation is still a bottleneck, sections 2.8 and 2.9 describe approaches to solving this problem.

Interpreters are frequently used to try out ideas in JVM implementation, in particular interpreters have been used to try out novel methods of object layout

and thread support.

In object layout, as highlighted in section 2.2, it is important to have compact data structures so as to be conservative on memory usage but also objects have to be easily supported by a fast garbage collection algorithm and also provide quick method dispatch. The Java class libraries place additional requirements on the object representation. The `java.lang.Class`, `java.lang.ClassLoader` and `java.lang.reflect.*` libraries allow programmers to access object internals and use the JVM's linking and loading abilities. A typical example of object layout trade-off was performed in the move from the Sun JDK 1.2 JVM [Sun99a] to the Sun HotSpot JVM [Sun99b] in the 1.3 release of the Sun JDK. In the 1.2 release objects required two indirections from an object reference to access a field. The first access was used to retrieve a handle, the use of handles vastly simplifies the garbage collector, the second access used the handle to access the field. In the HotSpot JVM handles were removed and only one indirection is required to access a field, this is at the expense of having a more complicated garbage collector.

All synchronisation in Java is performed on objects. Each object contains locking information and a set of threads waiting to get access to this object. Ideally the thread information will be small, the locking and unlocking primitives simple and the scheduler able to maximise the amount of time actually spent executing code on the processor rather than, for example, being stuck in a busy-wait loop waiting to acquire a lock to an object. The Java programming language makes it straightforward to put synchronisation checks on a region of code or even an entire method (the `synchronize` keyword), this ease of expression means that Java can suffer from thread overhead more so than other languages.

Thread optimisation also takes the form of mapping Java threads onto low-level operating system threads. This is the native threads model and is frequently complicated due to the difference between OS and Java threads. A green thread model maps the JVM's threads on to a scheduler within the JVM. The scheduler makes the multiple Java threads execute on the single thread which the JVM is running. Performance tends to be lower with a green thread model as the operating system can't allocate threads efficiently, for example, when waiting for device input or output.

## 2.8 Hardware JVMs

To solve the problem of bytecode interpreters being slow, hardware has been dedicated to the challenge of speeding it up. The PicoJava [MO98] processor is designed to execute bytecode natively and extends the bytecode instruction set<sup>1</sup> so that it can handle more hardware oriented tasks. The PicoJava processor has special support within its register file for the Java stack. The stack also imposes an extra overhead in that it requires operations to push and pop local variables on to it. Figure 2.6 shows an example of a RISC instruction and the equivalent bytecodes for adding two local variables together. It can be seen that the stack overhead is 3 instructions. PicoJava has a 4 instruction window in which it performs instruction folding, this removes the stack overhead and adds the 2 registers directly.

Java bytecode		RISC instruction	
iload_1	Push local 1 onto the stack	add r1, r2, r1	Add register 1 to register 2 and store in register 1
iload_2	Push local 2 onto the stack		
iadd	Add top 2 stack elements		
istore_1	Pop top of stack into local 1		

Figure 2.6: Comparison of bytecode and RISC

As well as dedicated Java processors such as PicoJava, ARM's ARM926EJ-S and ARM7EJ-S CPU cores have added bytecode as an extra operating mode for their processors [Cor00]. When in the bytecode mode the bytecodes are translated in the instruction decoder and issued to the processor like natively supported instructions. There are a great many restrictions on what bytecodes can actually be decoded on top of existing hardware. Bytecodes that are not properly supported trap out to handlers in the machines native instruction mode. This is inherently slower than direct support within the CPU, but it keeps the gate count of an implementation down.

Other processors that were designed with Java in mind but do not actually execute bytecode include the Delft-Java architecture [GV97, GV99] and the Jamaica architecture [WWEM99, WEMW99]. Delft-Java has a direct mapping of Java bytecode to its own RISC style ISA. To support the JVM stack the

---

<sup>1</sup>The last 53 bytecodes have no instruction assigned to them. This allows JVMs to extend the instruction set by either switching bytecodes when they are decoded or having library functions with these bytecodes in. An implementation which uses these bytecodes must ensure that none appear within a class file, as they pose a potential security risk.

Delft-Java ISA supports indirect addressing of registers. The Delft-Java uses a register renaming scheme to eliminate stack overhead. Jamaica uses its own light-weight thread mechanism to implement Java threads. Automatic parallelisation tools that increase the amount of parallelism within Java programs (such as JAVAR [BVG97]) are suited to the Jamaica architecture.

Java hardware has managed to achieve a speed up of 8x over interpreted JVMs [Cor00] but still are not as fast as dynamic compilers. By not recompiling code, hardware and interpreter JVMs save on memory. They therefore both have a niche in embedded applications where memory is limited.

## 2.9 Just-In-Time and dynamic compilation JVMs

Just-In-Time (JIT) compilation and dynamic compilation both refer to the method of executing Java bytecodes by translating them into native code and then executing the translated code instead of the bytecode. This causes a performance improvement as the translated code is cached and executed each time the bytecode should be executed.

A JIT compiler compiles the bytecode into native code either when a class is loaded or when a method is executed for the first time. A dynamic compiler is more selective over when compilation is performed and in certain cases a dynamic compiler may choose to interpret instead of compile the bytecode. This can hopefully avoid any unnecessary compilation expense. A dynamic compiler may also choose to optimise a hot region of code. The optimisation takes the form of conventional static compiler optimisations which, due to slowing the compilation process down, were not executed when the compiler was run the first time.

JIT compilation is a form of dynamic compilation where the heuristic of when to compile is very simple, once, when the class or method is first loaded or executed. The heuristic of when to compile can make all the difference with a JVM: compile too late and the benefit of faster code is never realised, compile too early and the dynamic compiler could compile code that will infrequently be executed. The dynamic compiler, like computer hardware, has to choose what will happen in the future from its past experience (for example, temporal locality properties exploited by caches). Java gives clues with methods such as class initialisers that they can only ever be run once. However, because a method is run once does not mean a lot of time can not be spent in it, for example, if there

was a large loop in the body of the method.

The optimisations to increase program speed performed by dynamic compilers are largely the same as in conventional static compilers<sup>2</sup>. Dynamic compilers have a benefit in that execution statistics of the code to compile are available as optimisations are performed. Chapter 4 describes the optimisations performed in the Dynamite dynamic binary translator, that is analogous to a dynamic compiler. As with Java hardware the overhead of a stack is eliminated by the dynamic compiler. Java bytecode allows a direct mapping of the stack to registers by making it a requirement of the bytecode that each time it is executed the stack depth be the same.

Section 1.2 introduced the problem that object-oriented programs tend to have short methods with lots of function calls and small basic blocks. A dynamic compiler will struggle to optimise a method as it has little code in it, performance will also suffer due to the overhead of passing information around inside the JVM. To counter this an optimisation called method inlining is performed.

### 2.9.1 Method inlining

Method inlining is a technique used in object-oriented environments to increase basic-block size and remove method boundaries therefore improving the opportunity for optimisation. A high-level example is shown in figure 2.7.

In figure 2.7 the code of class `test` has the method inlining optimisation performed to produce the class `test_optimised`. In the transformation the method body of `cube` is moved inside the method body of `main`. For the code to remain equivalent the parameter `x` is renamed `i`. This optimisation produces code that no longer needs to perform a method call and method return, it also does not need to pass `i` as a parameter. The body of the while loop in `main` is now longer, allowing techniques such as instruction scheduling greater scope for optimisation.

The low-level implementation of method inlining varies slightly from the example given in figure 2.7. In low-level implementations variables have symbolic names and do not need renaming. More general implementations of the method inlining optimisation need to be able to handle method polymorphism and in Java's case dynamic loading of classes. This means that an inlining optimisation may later prove to be invalid and the optimised code needs invalidating.

---

<sup>2</sup>Conventional static compiler optimisations in a dynamic binary translator environment are discussed in [Rog99].

```
class test
{
    public static void main (String args [])
    {
        int i=0, total=0;
        while (i < 100){
            total += cube(i);
            i++;
        }
    }
    private static int cube (int x)
    {
        return x * x * x;
    }
}

class test_optimised
{
    public static void main (String args [])
    {
        int i=0, total=0;
        while (i < 100){
            total += i * i * i;
            i++;
        }
    }
    private static int cube (int x)
    {
        return x * x * x;
    }
}
```

Figure 2.7: Example of method inlining

If the call site which is inlined into a method does call multiple locations it may still be worth placing the method bodies into the call site to remove the parameter passing and method return overhead. In this circumstance the inlined call site has to choose one of possibly several inlined methods or to perform a regular method call. In this situation, Class Hierarchy Analysis (CHA) can be used to determine whether a virtual call can actually be replaced by a static call. CHA shows success rates of between 14% and 40% at inlining virtual calls in Java [MMBC97].

The choice of which method to inline is guided by a weighting generated from the execution statistics of the caller and callee method, whether or not the called method lies within a loop, and what kind of a method is being compiled/optimised. Methods flagged as `final` in the Java program are prohibited from being overloaded, this is also true for private methods, these methods therefore make good candidates for being inlined as there can be no complication of the method dispatch.

One final heuristic for choosing a method for inlining is based on whether it is a leaf method or not. A leaf method is one that calls no other methods. A method can be flagged as a leaf method at the time it is loaded and verified by the primordial class loader. We discovered by instrumenting the Kaffe [Wil02] JVM that certain methods did benefit from this analysis. An example is the library method to read a string from a file, this method in turn calls a method to return a single character from a section of the file which is buffered. By inlining the call to read a single character the string reader loses the cost of a method call overhead as well as opening up the possibility for optimisation within the loop of the string reader method. In many ways this is similar to the example shown in figure 2.7.

## 2.10 HotSpot

HotSpot is a state-of-the-art JVM which executes Java bytecode using an interpreter, JIT and optimising compiler. JIT is used in this context to mean a simple non-optimising compiler. To switch between modes of execution synchronisation points are recorded in the code. At these points the JVM can switch from one mode to the other, Sun call this technique in-stack replacement.

Each of the HotSpot's 3 execution modes has different properties:

- interpreter: interpretation has very low latency meaning there is very little delay from the start of the code being told to run to it starting executing. The bytecodes instrument the execution of the method to guide the modes of execution that use dynamic compilation. HotSpot's interpreter is automatically generated at start-up using the inbuilt assembler used by the compilers. Generation is fast (less than 10ms on a 400MHz Intel Pentium II) and configured to the exact architecture the JVM will run on [Gri99]. The Spec JVM '98 benchmark [SPE98] score for the interpreter is 2.3.
- JIT compiler: originally this was a template-based compiler that translated each bytecode into a fixed set of native instructions. Template compilers have low latency but the performance of the produced code is slower than that of an optimising compiler. The next generation HotSpot JIT compiler performs optimisations such as common sub-expression elimination within extended basic blocks, simple register allocation based on usage counts, elimination of array bound checks in inner loops, method inlining and delay slot filling. This increased the complexity of the JIT more than 3 fold from 1400 cycles to translate a bytecode to 4300 cycles per bytecode on a Ultra SPARC system [AD00]. The performance of the translated code is more than 10 times faster than the interpreter with a Spec JVM '98 benchmark score of 23.6. The JIT compiler produces code which instruments the execution of the code to further guide the final optimising compiler.
- optimising compiler: this stage performs a full set of compiler optimisations and produces code based on execution statistics created in the previous two modes of execution. The compiler works in two phases, high form and low form. Optimisations performed during the high form phase of compilation include method inlining guided by class hierarchy analysis, common sub-expression elimination, dead code elimination and invariant code hoisting. During the low form phase optimisations include global register allocation, delay-slot filling and branch optimisation. A bottom-up rewrite system is used to convert machine-independent to machine-dependent instructions. This is guided by a deterministic finite state automata that records the lowest cost instruction as the intermediate representation is walked. During the final phase of code generation a peephole optimiser selects the best instruction to plant. Global code motion and null pointer check elimination

are also performed during optimisation [PVC01]. The performance of the final system is more than 12 times faster than the interpreter, but the number of cycles to translate a bytecode has increased to over 150,000 cycles [AD00]. The Spec JVM '98 benchmark score for the optimising compiler is 27.9 although this will increase as the compiler matures.

As some of the optimisations performed by the two compiler modes of execution may prove unsafe, a deoptimiser can remove optimisations. The main optimisation to be affected is method inlining. Method inlining is potentially made unsafe whenever a new class is loaded. The deoptimiser has to recreate the state necessary for the interpreter mode of execution. To do this safe points are planted through out the generated code and when one of these is reached deoptimisation is performed.

Two versions of HotSpot exist for the server and client markets. The server JVM focuses on high performance whilst the client JVM focuses on low latency. By tuning when each mode of execution is used the different server and client configurations are realised. The server JVM tries to switch to the dynamic compilation phase as early as possible, where as the client JVM will usually operate in the JIT mode. As the JIT mode of execution is only around 20% slower than the optimising compiler mode, most client processes do not notice the speed loss and benefit from the decreased latency caused by faster translation. As server applications tend to be larger and run for a much longer period of time, the time spent in dynamic compilation gets returned by the faster execution of the code. In client applications, the cost of dynamic compilation may never get returned through increased performance, so the compilation process is executed on a low priority thread.

## 2.11 Summary

This chapter has introduced Java and the components that make it up, differentiating between the Java programming language, the Java Virtual Machine and the class libraries, and introduced the Java environment. The chapter went on to describe key features of Java to the JVM: object representation, the class file format and dynamic link loading. Next different implementations of the JVM were considered from interpreted, hardware and dynamic compilation. Dynamic compilation offers the highest speed of execution with expensive optimisations

such as method inlining being performed, however, this can cause the JVM to suffer from latency problems. Finally, the HotSpot JVM was described as well as its mechanisms for dealing with latency.

More generally, this chapter has described techniques used by other Java and JVM implementations such as IBM's Jalapeño [AAea00], MS Visual J++ [Mic01a], Insignia's Jeode [Sol01], Symantec [Nic98], Cocoa [KG97,Kra98], Caffeine [HGH96], Kaffe [Wil02], OpenJIT [MOS<sup>+</sup>98], Toba [PTB<sup>+</sup>97], SableVM [GH01] and TowerJ [Tow98].

Chapter 5 goes on to describe the Dynamite JVM that implements a complete JVM on top of the Dynamite dynamic binary translator. The following two chapters describe dynamic binary translation technology.

# Chapter 3

## Dynamic Binary Translation

This chapter describes dynamic binary translators and is followed by chapter 4 describing the Dynamite dynamic binary translator on which the Dynamite JVM is built. The main part of the chapter comprises three sections (3.2, 3.3 and 3.4), highlighting certain binary translators. The next section provides an overview of binary translators and the three groups used to classify binary translators in this thesis.

### 3.1 Overview

Dynamic binary translation is a technique used to run programs that are not natively compatible with a computer platform<sup>1</sup>. Binary programs have little structural information contained within them and are sometimes impossible to reverse engineer, thereby inhibiting migration to a new computer platform.

For example, code compiled for an Intel IA-32 machine, running Microsoft Windows, has information about functions and subroutines in the source code at the time of compilation. The compiler generates binary code that has data structures placed in with the code, the data typically being used for jump tables and other precomputed values. The compiler or the programmer may even create code that is self modifying (for example, value specific optimisation [Kep96]). A static binary translator can only translate the effect of this data and code if it knows the format in which it was generated. This may alter as computer

---

<sup>1</sup>Dynamic binary translation is one form of computer emulation. Emulation is typically meant to mean a emulated environment built around an interpreter. For this reason dynamic binary translators try to distinguish themselves from emulators.

languages and compilers change; hand-crafted assembler code has no style to conform to.

Dynamic Binary Translators (DBTs) do not suffer from problems of code discovery, as they translate and execute code in a lazy manner. Only code used within a program's execution needs to be translated. If a previously untranslated path is taken by a branch then the DBT is called to translate the path. The new path taken is calculated by the executing dynamically generated code.

The term *subject machine* is used to refer to the computer architecture a program was intended to run on. The *subject environment* is the environment the DBT produces. This may be little more than the subject machine, but it may include loaders, linkers, and emulated software calls or hardware devices. The *subject program* is the program the subject environment executes. The instructions contained in the subject program are *subject code*. Likewise, the code produced by the DBT is called *target code* and is executed in a *target environment*.

In 1987 Hewlett-Packard released a binary translation system with a HP 3000 subject machine and a PA-RISC target machine [AKS00]. Later, Tandem built a translator and interpreter system to migrate users of legacy TNS products to their newer TNS/R product line [AS92]. Dynamic Binary Translation is now a key technology allowing a computer architect to be freed from legacy code concerns. Table 3.1 highlights emulation and DBT environments of the last 15 years.

Tool	Reference	Function
OCT + HP 3000 Emu- lator	[BKMM87]	Static translator (Object Code Translator - OCT) and interpreting emulator to run HP 3000 MPE V binaries on PA-RISC MPE XL.
Flashport	[GM88]	Static translator with configurable subject environment front-end and target environment backend modules. Code discovery relies on human input. 680x0 Macintosh, IBM 360/370/390 and PDP11/70 subject machines supported. PowerPC Macintosh, IBM RS/6000 and PowerPC, SPARC SunOS, PA-RISC, MIPS and IA-32 target machines supported.
XDOS	[HB89]	Static translator of IA-32 MS-DOS binaries to 680x0, IA-32, SPARC, 88000 and MIPS Unix. Human intervention for code discovery like Flashport.

Bedichek	[Bed90]	Fast DBT based 88000 architecture simulator with debugger interface that runs on a 68020.
Accelerator	[AS92]	Static translator and interpreter to run TNS CISC binaries on TNS/R.
VEST & TIE	[SCK <sup>+</sup> 92, SCK <sup>+</sup> 93]	Static translator (VEST) and runtime environment with an interpreter (TIE), to run VAX OpenVMS binaries on Alpha OpenVMS.
mx & mxr	[SCK <sup>+</sup> 92, SCK <sup>+</sup> 93]	Static translator (mx) and runtime environment with an interpreter (mxr), to run Ultrix MIPS binaries on Alpha OSF/1.
Mae	[App96]	68LC040 Macintosh emulator for SPARC Solaris and PA-RISC HP-UX.
Wabi	[Cal02]	IA-32 Windows API subset on Sparc Solaris, PowerPC AIX and IA-32 Linux.
Atom	[SE94]	Binary modification tool for instrumenting binaries and collecting extra profiling statistics.
Shade	[CK94]	DBT that translates MIPS or SPARC code to run on SPARC SunOS/Solaris. Translated code is augmented to generate profiling information.
Executor and Syn68k	[Hos95]	Efficient interpreter emulating 68LC040 on IA-32, 680x0, i860 and Alpha.
TIBBIT	[CS95]	68000 to C to PowerPC with a focus on maintaining real-time constraints.
SoftWindows and RealPC	[Sof02]	IA-32 hardware emulation using an interpreter and DBT targeting PowerPC Mac OS and SPARC, PowerPC and MIPS Solaris.
Virtual PC	[Tra97]	IA-32 hardware emulation and DBT to PowerPC Mac OS.
Freeport Express	[Com02]	SPARC SunOS to Alpha Digital Unix static translator (fpx) and runtime environment (fpxr) with a similar execution model to VEST and mx.
SimOS	[RBDH97]	High-performance configurable architecture simulator similar to Shade.
Embra	[WR96]	Complete emulation of MIPS R3000/R4000 using SimOS's DBT.
Morph	[CSB96]	Binary migration tool supported through compiler and measurement tools that annotate binaries.
FX!32	[Har97]	DBT to run IA-32 Win32 binaries on Alpha Windows.

DAISY	[EA97]	VLIW architecture with extensions supporting binaries from different instruction set architectures using a DBT called the VMM.
Bochs	[Boc01]	Unix IA-32 Emulator with a limited and now discontinued DBT.
Crusoe	[Kla00, KCW98]	VLIW hardware dedicated to the fast dynamic translation and execution of IA-32 instructions.
Dynamo	[BDB99]	DBT and optimiser running PA-RISC HP-UX binaries on PA-RISC HP-UX.
VMWare	[VMw99, Kei01]	IA-32 hardware emulation on IA-32 Windows and Linux with instructions running natively.
MOJO	[CLCG00]	DBT and optimiser to run IA-32 Win32 binaries on IA-32 Win32, similar to Dynamo.
Plex86	[Law00]	Open source DBT project to translate IA-32 to multiple target machines.
UQDBT	[UC00a]	Generic DBT built using machine descriptions. Runs IA-32 or Sparc Solaris binaries on IA-32 or Sparc Solaris.
Aries	[ZT00]	DBT to run PA-RISC HP-UX binaries on IA-64 HP-UX.
Vulcan	[SEV01]	Runtime system providing feedback to optimise binaries across platforms.

Table 3.1: Translation environments

Figure 3.1 shows three different combinations of subject and target environments, with the DBT program layer.

In this thesis type 1 DBTs will be referred to as above operating system DBTs, type 2 as between operating systems DBTs and type 3 as below operating system DBTs. In this chapter above operating system DBTs are described in section 3.2, between operating system DBTs in section 3.3 and below operating system DBTs in section 3.4.

As all DBTs are performing a similar role they may be adapted, for example, from a between operating system DBT into a above operating system DBT. What alters in this transition is the support for the surrounding operating system and hardware.

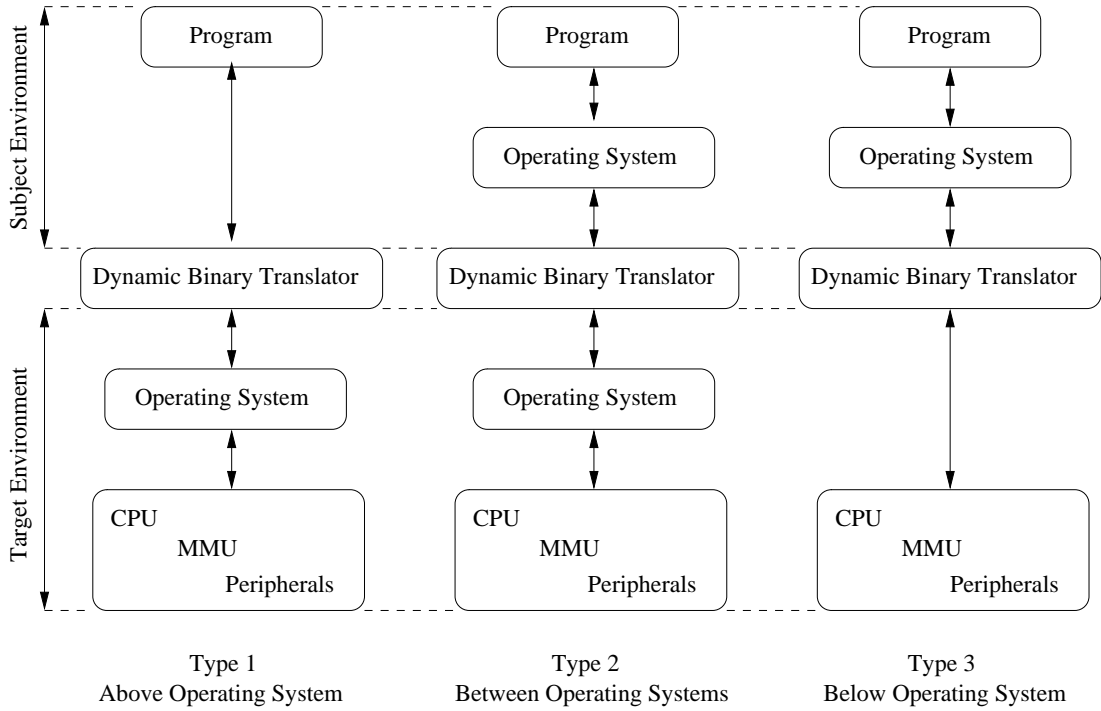


Figure 3.1: Dynamic binary translation environments

## 3.2 Above operating system dynamic binary translators

Above operating system DBTs take a program and execute it directly on the DBT. As the underlying operating system is not translated, system calls need to be passed through. Calls that cannot be passed need to be emulated by the DBT. FX!32, Dynamo, UQDBT and DAISY all operate in this manner and are described respectively in sections 3.2.1, 3.2.2, 3.2.3 and 3.2.4.

### 3.2.1 FX!32

In the early 1990s Digital Equipment Corporations ambitiously planned to migrate Windows NT, VAX/VMS and MIPS/Sparc Unix users to its new Alpha processor. The Alpha was, and still is, a very fast RISC processor that looked as though it could take over from existing CISC based processors, such as the Intel i486. Microsoft offered support to the new architecture by porting their Windows NT platform [Mic01b].

To enable people to switch easily to the new platform, Digital produced

FX!32 [Tur96,HH97]. This DBT allows Intel IA-32 Microsoft Win32 programs [Har97] to run on the Alpha version of Windows NT. A Win32 binary is a binary of a program designed to run on Windows NT and also Microsoft's later operating system product line of Windows 95, 98, etc. A Win16 binary designed for early Microsoft operating systems, such as Windows 3.1, would work through an emulation environment that interpreted rather than translated IA-32 instructions.

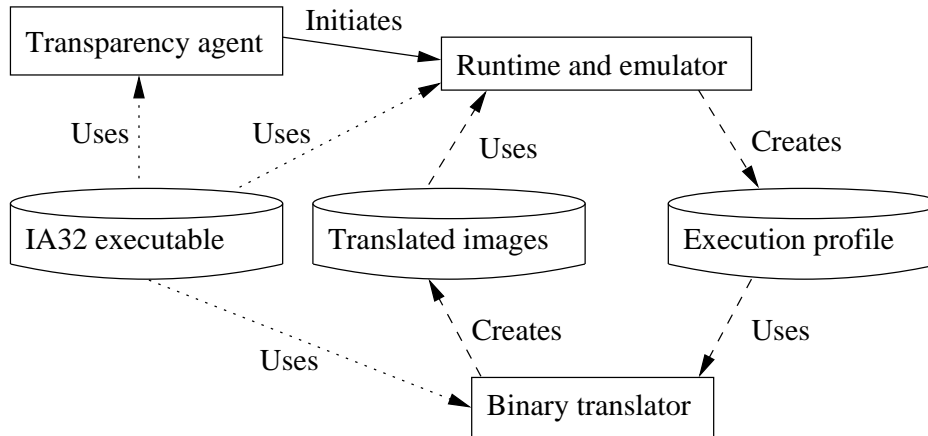


Figure 3.2: The FX!32 System

The FX!32 system is shown in figure 3.2. The transparency agent is linked into the Windows NT operating system and handles the loading of the runtime environment for executing the non-native code. The translator itself is run as a low-priority task. Sections of IA-32 programs are translated, based on execution profiles that are gathered by the operating system. The profiles determine which binaries should be translated first. The runtime environment and emulator use a translated image, if one exists, or as a fallback for a half translated binary, the system interprets the IA-32 instructions.

FX!32 incorporates a number of optimisations novel to its purpose [HH97]. The translator has two mangler functions for handling IA-32 registers and flags (condition codes). The register mangler maintains the contents of the IA-32 registers in separate 32bit registers. The IA-32 Instruction Set Architecture (ISA) defines registers such as EAX, AX, AL and AH to be overlapping, so the mangler has to recombine registers whenever appropriate. By only recombining registers when necessary, FX!32 avoids frequent shifting and masking of registers. The flag mangler avoids maintaining the status of the flags during a translation, by generating the flags when necessary from saved state at the point the flags are used. As the values in the flag registers are frequently over-written, the flag

mangler need only store state information for the end of a block. FX!32 handles through profile analysis the problem of the Alpha processor faulting on unaligned memory accesses. When profile data shows unaligned memory accesses to be common, different code is generated which faults less frequently.

Intel and AMD provided stiff competition to Digital, with the Alpha processor finding a niche away from the desktop in the high-performance server market. FX!32 has now been discontinued, the Alpha processor (licensed to companies such as Samsung) is used in systems such as the Cray T3E [Cra01].

### 3.2.2 Dynamo

HP started the Dynamo [BDB99] project in 1995 to create a DBT of a speed and transparency that is comparable to how applications are run natively. Initial Dynamo environments translated from PA-RISC HP-UX to PA-RISC HP-UX and so are not translators as such, but the execution model supports different front-end modules being written to configure the subject environment. Figure 3.3 shows the execution model of Dynamo.

Dynamo starts off by interpreting subject machine instructions on a model of the machine's state held in a context. On the PA-RISC platform (as the subject and target environments are the same), the interpreter can be replaced by running the native code and then performing a trap out when a start of trace point is reached. An example of a trace start point would be a backward branch. However, trapping out of code is reliant on a light weight trap instruction. When a trace start point is hit, the Dynamo system looks to see if the proceeding code has been previously translated. If it has, then this is executed until an exit point returns the system to the interpreter environment with the context updated. The translator is called when a section of code, pointed to by the program counter, is detected as being hot (frequently executed). The translator interprets the instructions and builds a trace profile until an end of trace condition is reached. This profile is then translated, optimised and placed in the cache of translated code. Linking the code in the code cache avoids the cost of interpretation and detecting whether or not a program counter has been translated previously.

Key to Dynamo's optimisation strategy is trace scheduling. The traces that are built are ordered so that the most frequently executed paths experience reduced branch and cache misprediction penalties. Dynamo also concentrates on being small and having a small code cache. The small size of Dynamo in memory

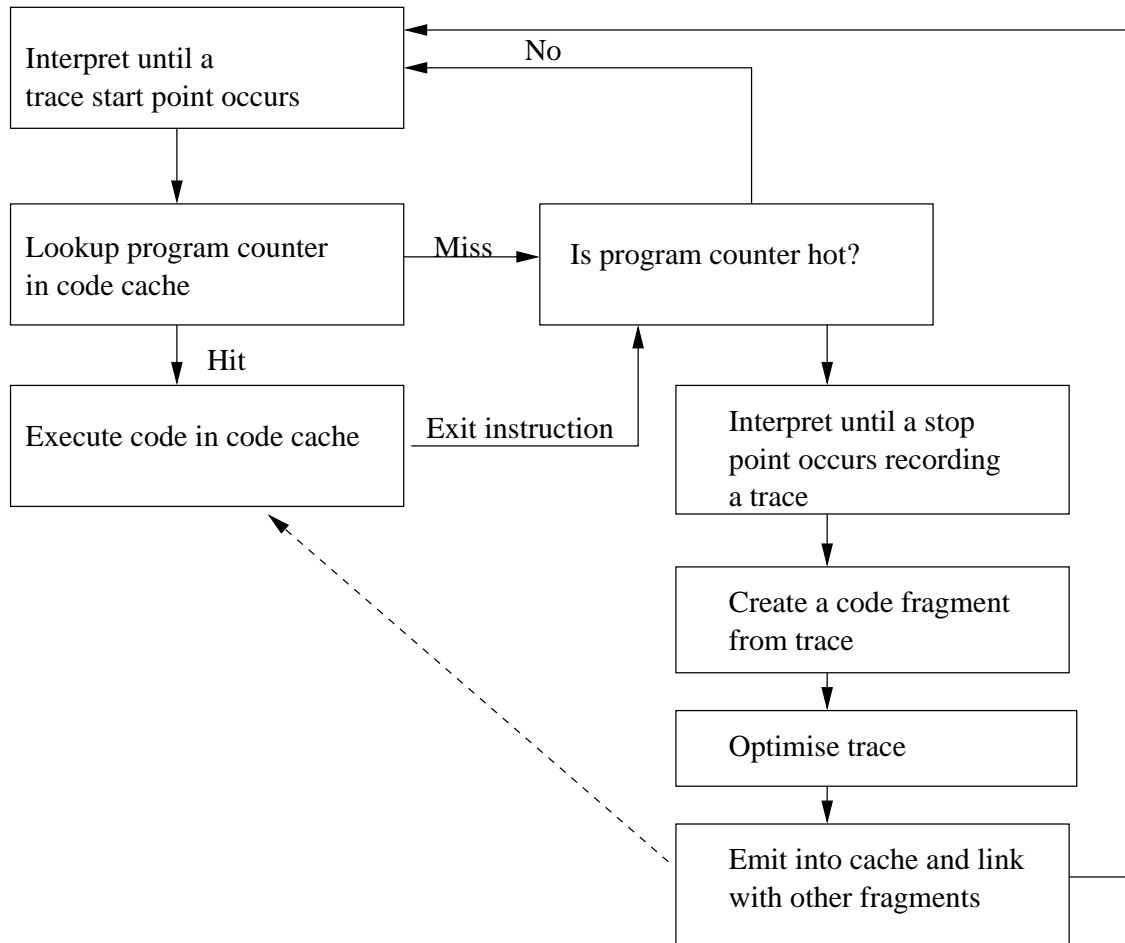


Figure 3.3: Dynamo model of execution

comes from focusing on optimising a simple execution environment, with straightforward heuristics. The small code cache size comes from frequently flushing the code cache whenever a change is detected in the way a program is executing (referred to as program behaviour).

On the suite of SpecInt 95 benchmarks [SPE00] dynamo has a performance improvement equivalent to -O4 optimisations from their HPUNIX compiler run with -O optimisations [BDB00]. On certain benchmarks it does, however, perform worse than executing the program natively.

### 3.2.3 UQDBT

The University of Queensland Dynamic Binary Translator (UQDBT) [UC00a] builds on a static binary translator called UQBT [CE00]. A key part of UQBT's

static translation is the rediscovery of high-level programming language constructs such as switch statements [CE99] and the recovery of procedural information [CS00]. By reconstructing high-level programming structures the translator is able to use an appropriate target machine's mechanism to implement the translation. For example, C code compiled for the IA-32 ISA [Int87] passes parameters to procedures through the stack, whereas the SPARC ISA [SPA92] uses a register-window scheme. The UQBT translator can map the SPARC scheme onto the IA-32 scheme or vice versa, whereas a conventional translation system would have had to emulate each scheme.

UQDBT's optimisations work by inspecting code and interpreting it to give certain properties for the code. This is not always possible as: functions that take a variable number of arguments have complicated function calls, hand coded assembler can disregard convention, a compiler's optimiser may remove part of the convention as part of their own optimisation scheme or the code could contain self-modifying portions.

UQDBT uses mechanisms developed for UQBT such as subject and target machine specification languages to provide machine adaptability. UQDBT does not use any procedure discovery mechanisms, as in UQBT, because this optimisation proves too costly to be executed at runtime. Instead cheap analysis is performed and optimisations used on frequently executed regions of code.

UQDBT translates and caches basic blocks, unlike Dynamo which translates traces of multiple basic blocks. A basic block is a small fragment of code (described further in section 4.2.1), so it is necessary for UQDBT to build larger fragments to avoid the penalty of being in the translator's code instead of the dynamically generated code. UQDBT builds up hot paths [UC00b], which are collections of basic blocks connected by edges that have been traversed a number of times greater than a threshold value. Hot paths are analogous to group blocks in the Dynamite DBT described in chapter 4.

UQDBT maps as many system and library calls across architectures as possible. In particular, UQDBT concentrates on running IA-32 Solaris binaries on SPARC Solaris. The main problem in achieving this is that SPARC is a big-endian architecture, whilst IA-32 is a little-endian architecture. Byte swapping and memory alignment prove a performance bottleneck at the library interface.

### 3.2.4 DAISY

DAISY stands for the Dynamically Architected Instruction Set from Yorktown [EA97]. The DAISY project has at its heart a novel instruction-level parallel machine and VLIW ISA. To gain acceptance, DAISY uses a DBT called the Virtual Machine Monitor (VMM). The subject environment emulated by the DAISY DBT was primarily the Power PC ISA, however, the ability to configure the subject machine is designed into the translator.

DAISY initially uses an interpreter to interpret code pieces and gather profile data. A code piece is interpreted up to an execution threshold, at which point it is translated. The translator builds up tree groups, which pass control between each other or back to the VMM [EASG99, EAGS00]. In the original DAISY implementation, branches that used a register or branches across pages were used to trigger the VMM. A page fault was generated when an untranslated page was branched to, this provided a trap into the VMM. A tree group is analogous to a hot path in UQDBT, except that profile information is not held on branch direction (counts associated with arcs between addresses), but on the individual blocks themselves (counts associated with a particular address).

The DAISY architecture includes innovations that enable it to reschedule loads and stores, and also accurately and quickly generate exceptions. DAISY achieves this by having an exception bit in a number of special registers. The VMM generates code that uses these registers. If an exception occurs on an instruction accessing one of these registers then the exception bit is set, but the exception is not reported to the operating system. If a register with the exception bit set is copied to a regular register then an exception is thrown.

DAISY's support of dynamic compilation made it a good candidate to implement a JVM [EAH97, YMP<sup>+</sup>99]. The JVM implementation techniques used are described in chapter 2.

## 3.3 Between operating system dynamic binary translators

Running one operating system within another is commercially popular. A number of such emulation environments exist, with a few using translator or DBT technology. Sections 3.3.2, 3.3.1, 3.3.3 and 3.3.4 describe four such environments.

### 3.3.1 VEST and mx

VEST and TIE are tools to translate and run VAX OpenVMS binaries on Alpha OpenVMS [SCK<sup>+</sup>92,SCK<sup>+</sup>93]. Mx and mxr are tools to translate and run MIPS Ultrix binaries on Alpha OSF/1. The execution process of these two systems is shown in figure 3.4.

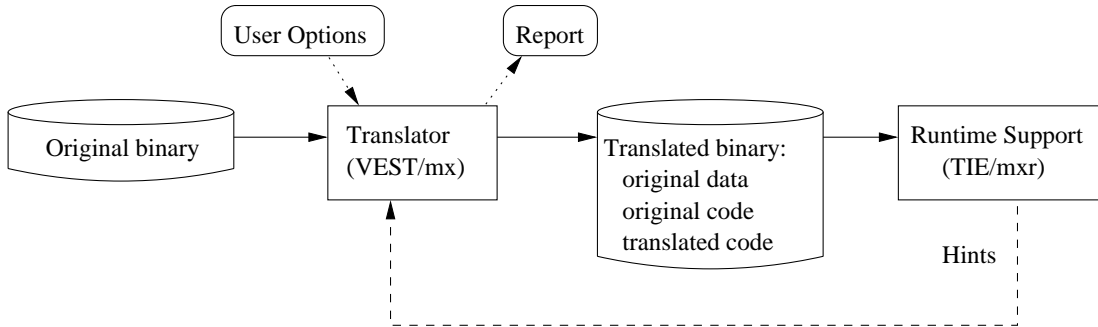


Figure 3.4: VEST and mx execution model

TIE/mxr interprets any instructions that VEST/mx has not been able to discover, as well as providing the emulation of the software environment. The interpreter generates a hints file to aid in code discovery if the VEST/mx system is re-run. The VEST/mx translator builds up a flow graph of basic blocks within the program. Branches are resolved in this process using symbolic execution of instructions.

VEST and mx optimise the translated code by use of idiom recognition, whereby a sequence of instructions is replaced by a more optimal one for the target machine. They also perform instruction scheduling to optimise the translated binaries performance on the Alpha processor. A key concern for these systems is to be 100% architecture compatible with the environment they emulate. This puts a burden on the translator not to alter exception handling behaviour. This is complicated in the case of VEST, which must translate the VAX CISC instructions into multiple Alpha RISC instructions.

### 3.3.2 Macintosh application environment

The Macintosh Application Environment (MAE) [App94,App96] provides an emulation of the 68LC040 and Macintosh hardware for Solaris and HP-UX. MAE offers users of Solaris and HP-UX the ability to run MacOS software, and also

offers administrators a mechanism by which they can centralise administration for multiple virtual machines on one high-end server.

MAE's control loop uses a lookup table of 68LC040 instructions to interpret each subject instruction. Hot-block detection is performed, the control loop branches into hot regions of code by planting the address of the translated code in the lookup table and altering the subject machine's instructions so that it branches to it. Hot-blocks are determined using a sampling algorithm. The start of the hot-block is a branch instruction and hot-blocks are ended by non-local returns, non-local unconditional branches and complex instructions.

A legacy system to enable PowerPC Apple computers to run 68000 series programs works in much the same way as MAE except it does not have to provide an emulation environment, it can just pass through API calls to the operating system kernel. It also has different requirements placed on its memory footprint.

MAE has now been discontinued by Apple.

### 3.3.3 Wabi

Wabi [Cal02] is a DBT that runs IA-32 applications on SPARC Solaris and PPC AIX. More recently Wabi has been dropped by Sun and sold to Caldera who have ported it to IA-32 Linux. The IA-32 version does not perform any translation, instead running the IA-32 instructions natively. API mapping is done from the Windows *create windows* API to the X-Windowing system *create windows* API. The API is incomplete so only a subset of MS Windows programs are compatible with Wabi, 24 applications being directly supported.

Sun produced the IA-32 to SPARC DBT for Wabi whilst IBM used its 80x86 Instruction Set Translator for the PPC. Both perform code analysis and caching of hot regions, to help improve performance. The SPARC DBT maps IA-32 flags onto SPARC flags, eliminating any unnecessary updates by evaluating the flags as late as possible.

### 3.3.4 VMWare

Sometimes the motivation of an emulator is to run one operating system inside another, on the same ISA. In such cases, code does not need translating and can be emulated in a separate memory space. VMWare [VMw02] provides such functionality allowing an IA-32 based PC to be emulated within itself. VMWare

has to provide an emulation of the PC's hardware and BIOS with the emulated PC appearing indistinguishable to the guest operating system. SoftWindows [Sof02], Bochs [Boc01] and Virtual PC [Tra97] also emulate a full set of PC hardware.

Network service providers are using full PC emulators due to their ability to emulate network cards and appear as extra machines on a network. By providing customers with virtual machines costs can be kept low, through lower space rental and cost of hardware, whilst keeping a stable system. Stability is provided by each virtual machine running in its own address space.

## 3.4 Below operating system dynamic binary translators

Type 3 DBTs execute a full operating system and program straight on an underlying piece of hardware. The DBT needs to map hardware or emulate it. Not having an underlying operating system means the DBT has to provide its own support libraries. Running directly on hardware gives a DBT the greatest amount of freedom in optimising its code.

Simple DBTs, like simple JVMs, can interpret a subject machine instruction on the target machine using a loop and state variables [AS92]. Alternatively, there can be microcoded solutions such as the Intel 80286 compatability on 80386 and later processors [Int87].

### 3.4.1 Transmeta

The Transmeta Crusoe processors provide compatability with Intel IA-32 instructions at the same time being a small, low powered processor with a native VLIW instruction set [Kla00]. The Crusoe processors move complexity out of the instruction decoder and into the code morphing software, which is a DBT. They use a special translator memory to hold the DBTs translations. Enhancements are not only brought by new silicon but improved by new software too. For example, version 4.2 of the code morphing software achieves up to a 40% reduction in power consumption on benchmarks, compared to version 4.1 [Tra01].

The Crusoe processors incorporate features to enable rollback of several instructions, allowing portions of code to be aggressively optimised and for quick

recovery of state if these optimisations are invalid [KCW98]. When an optimisation violation is detected the DBT is called to retranslate the code more conservatively. The rollback mechanism is implemented through a gated store buffer and additional registers. The gated store buffer delays writes to memory until a boundary of a region of code is reached. If a fault occurs the memory state is no different than when the region was entered. The registers are copied to and then switched for an alternate set, at the boundary. The none working set of registers preserves the register values from the beginning of the block. The mechanism resembles hardware-based speculation [HP96] where mispredicted instruction traces are discarded. The granularity is much coarser here as the DBT is generating code for multiple basic blocks. The DBT avoids paying the penalty of speculating incorrectly each time the code is executed, by retranslating the code and replacing the old version in the cache.

To detect a memory alias the Crusoe processor performs a load and protect instruction. This instruction loads a value and places the address it was loaded from into a special register. Loads and stores occurring after the load and protect instruction generate an address which is compared with that held in the load and protect instruction's special register. If the two addresses are identical then a fault is generated, the processor rolls back, and the DBT is called to retranslate the code.

Crusoe hardware is designed to target the low power market segment, such as laptops, embedded devices, and high-density servers. Performance is comparable with other processors in this market segment, but not comparable to high-end server and workstation processors.

### 3.5 Other directions

Above the operating system DBTs are well positioned to take advantage of binary format, Application Binary Interface (ABI) and library standards, developed by groups such as the open-group [Gro02]. iBCS2 is one such standard that defines linking, ABI, and libraries, that allow IA-32 compiled binaries to be run under many different operating systems. With dynamic binary translators, iBCS2 becomes a platform independent binary format, in much the same way as the Java class file.

As well as running application programs on top of operating systems, the

embedded computer market is a candidate for DBT technology. Embedded computers provide a wide range of functionality on different CPU cores. Porting code from one system to another can prove slow, as embedded code can have a lot of hand-crafted assembler code. Porting code becomes increasingly complicated when the systems have different word sizes and byte sex.

Peripheral devices, such as graphic cards, can off-load some of their initialisation on to the host processor. Typically, the host processor is expected to be IA-32 compatible. To enable these devices to work on machines that do not have an IA-32 compatible host processor, the BIOS (or even the graphic driver on machines with more than one graphic card) interprets the initialisation code. Devices such as modems have off-loaded some of their functions onto the host processor too [Hac99]. DBTs can enable these devices to run on alternate processors and operating systems.

As well as using a DBT as an emulator, it is also possible to use them to support software and hardware development, by translating and augmenting code. One typical augmentation is for generating statistics on how many instructions a program runs. Having a short execution time is not a key for these DBTs, as they will only be used by developers. The Shade tools [CK94] and SimOS [RBDH97] are in this category of DBT.

Finally, DBTs can be seen as part of a system that follows software through its life [SEV01]. Parts of a software's life that can be handled by a DBT include software management, anti-piracy, reliability, testing, performance measurement and optimisation. Software management is the process of delivering, installing and patching programs. Anti-piracy can be supported in a DBT by obfuscating a program's symbols and instructions, and detecting software tampering by the addition of watermarks to software. Reliability can be measured, as a running system can have tests run on it to verify it is operating correctly. Testing support can be added to an execution environment so that software can be tested in a live environment. Feedback about how the software is used can be analysed to guide features for later revisions, as well as spot bugs and monitor performance. DBTs provide optimisation, but within a fuller compiler and execution environment within a system with more feedback, then the optimisations can be better targeted and the performance further improved.

## 3.6 Summary

This chapter has introduced existing dynamic binary translator environments providing context for the Dynamite DBT. Dynamic binary translators have been used as:

- A migration tool to move users from an old ISA to a new one.
- Quick hardware emulation tools and statistics generator.
- Integral part of a high-performance, low-power CPU to reduce hardware complexity.
- Potentially future compiler toolkits, featuring DBTs, may be used as an integral part of software development.

The following chapter describes the Dynamite DBT in detail. The Dynamite DBT is the foundation for the Dynamite JVM, and all the following work in this thesis.

# Chapter 4

## Dynamite

*In the original thesis, this chapter provided background information, concerning the Dynamite binary translation system, which was necessary to understand the technical detail contained in subsequent chapters. However, this version has been abridged to avoid possible disclosure of information which is the intellectual property of Transitive Limited. Brief explanations of the relevant features are provided with references to publicly available material which contains sufficient information.*

*The author regrets the additional difficulty which this adds to the reading of the thesis.*

Research undertaken at the University of Manchester and now at Transitive Corporation has produced the Dynamite DBT [Sou96]. The subject environment is configured by a front-end module and code for a target environment is produced by a back-end module. Figure 4.1 shows the modules within the Dynamite DBT.

This chapter describes the four modules of Dynamite as well as optimisations performed within the framework.

### 4.1 The Dynamite fuse module

The fuse module performs the task of mapping between different operating systems and/or hardware. Details of this can be found in BNWK03, an overview is provided on the Transitive website Cor05.

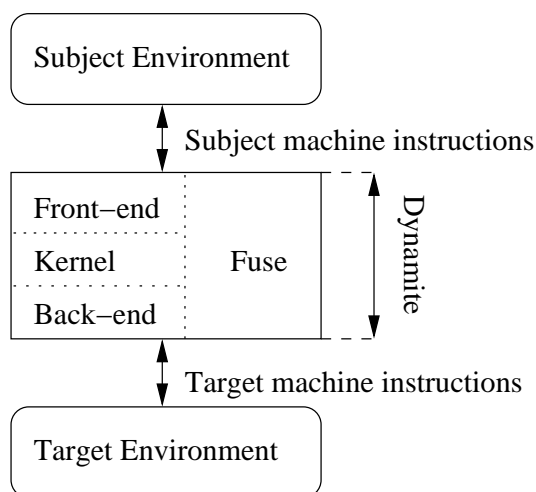


Figure 4.1: Dynamite modules

## 4.2 The Dynamite front-end module

Dynamite uses a front-end to build up an intermediate representation (IR) of the subject machine's instructions. Details of this can be found in RSJ<sup>+</sup>03, an overview is provided on the Transitive website Cor05.

### 4.2.1 Building Dynamite's intermediate representation

Descriptions of the IR are contained in RSJ<sup>+</sup>03.

The Expression IR was designed to be translated into VCODE input [Eng96]. VCODE is a retargetable back-end interface with its input being code that resembles a RISC load/store architecture.

The Dynamite front-end translates a series of subject machine instructions up to a control-of-flow instruction such as a branch or subroutine call. Aho, Sethi and Ullman [ASU86] define a basic block as follows:

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Dynamite's definition of a basic block is slightly different as it takes into account a variety of optimisations and special cases [SN03].

### 4.2.2 Substitute calls

To allow an interface to software and hardware, calls can be created as IR nodes. Details on this process can be found in BNWK03, OAHH03a.

## 4.3 The Dynamite kernel module

The Dynamite kernel comprises the generic parts of the Dynamite DBT framework. The intermediate representation forms part of the kernel, as does the code to translate it. The translation modules are tuned to the back-end using back-end configuration files.

### 4.3.1 Control

The control and execution of Dynamite are described in Nor04.

Dynamite is made up of a number of modules, a simplified view of the modules is shown in figure 4.2.

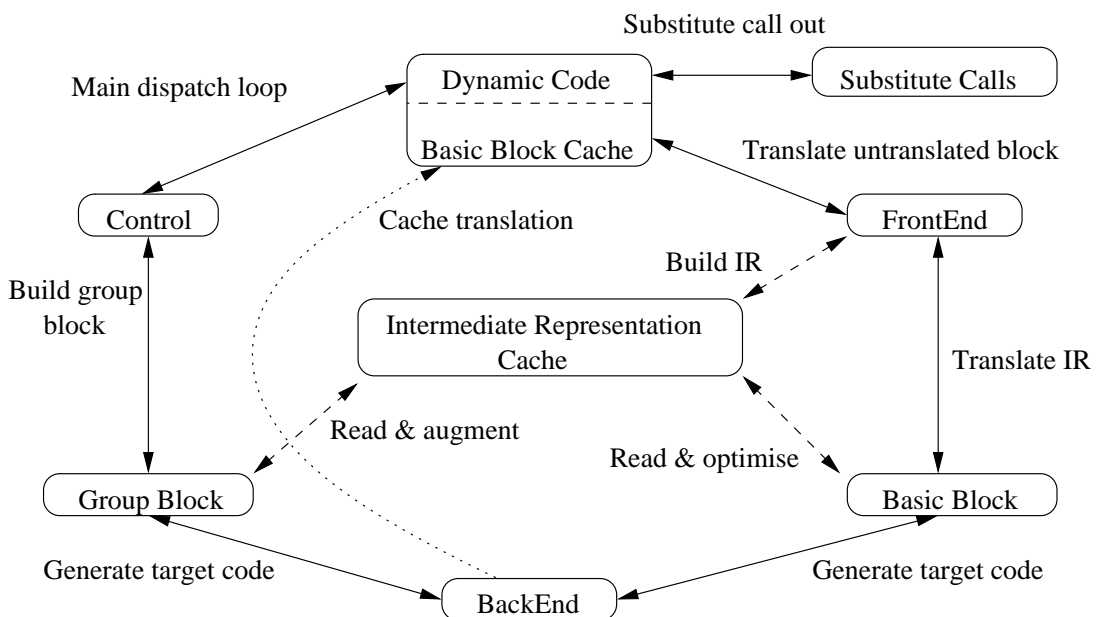


Figure 4.2: Interaction of Dynamite modules

After initialisation execution is passed to the control module. The control module consists of a loop that dispatches translated code, optimising the cached translations when appropriate. The optimised unit is called a *group block* and

resembles the larger blocks of compilation such as the trace scheduled block in Dynamo, hot paths in UQDBT and tree groups in DAISY. Group block construction is described in section 4.3.4. Chapter 3 described optimised regions for other DBTs.

The basic block cache, described in section 4.3.2, maintains the translated code and is responsible for initiating a translation. When a translation is required the front-end is called to translate a block and build up IR as described in section 4.2.1. The translated IR is cached as it will be used to translate the basic block as well as, later, to build the group block. The back-end is used by both the basic block translator and group block translator and is described in section 4.4.

Initially Dynamite executes translations of basic blocks, gathering execution statistics. When the execution count of a basic block is high enough group block formation and translation is initiated.

### 4.3.2 Basic block cache

As can be seen in figure 4.2 the basic block cache is a key constituent of the Dynamite kernel. It is the repository used for storing basic blocks.

Since statistically 90% of execution time is spent in 10% of code [HP96], the basic block cache can remove infrequently executed code and IR as there is only a slight risk it will need retranslation later. This assumption follows a pattern commonly made by computer architects that future execution will reflect current execution patterns. It is possible to think of cases where this will not hold and the basic block cache may need tuning so that it does not throw away these translations. For example, basic blocks associated with certain subject memory locations, like library routines, should not be removed.

As described in section 4.2.1 basic blocks in Dynamite can overlap due to their discovery mechanism. Dynamite also allows specialisation of basic blocks as described in section 4.3.3. This leads the execution statistics of how many times a basic block has been executed not to be identical to the number of times a region of code is executed.

### 4.3.3 Basic block compatibility

Basic blocks are allowed to branch to other compatible basic blocks via a compatibility test. SN03 provides details on how this is used and may be used for

optimisation. This optimisation is similar to the register mangling performed by FX!32 (described in section 3.2.1).

#### 4.3.4 Group blocks

When a basic block is executed over a trigger threshold a group block is formed. Group blocks have a reduced translator overhead as they no longer need to enter the control module between basic blocks. A description of group blocks is contained in RSJ<sup>+</sup>03.

#### 4.3.5 Dead code elimination

The dead code elimination technique for Dynamite is presented in RSJ<sup>+</sup>03.

#### 4.3.6 Constant propagation

The constant propagation technique for Dynamite is presented in RSJ<sup>+</sup>03.

#### 4.3.7 Value-specific optimisation

The latency of certain instructions inside a CPU can lead to pipelines being starved of work. Lipasti and Shen [LS96] suggest a microarchitecture device that can speculate on the values returned by loads. The load would take place and if the speculation proved correct then the results would be committed. Prediction bits tell the instruction decoder whether to speculate on the value of a particular load, these bits are updated depending on the results of prior predictions, this has a lot of similarities with branch prediction. As the pipelines would have had to stall for the load operation the CPU has not wasted any clock cycles. The disadvantage of this optimisation is that it is expensive in CPU area to make a value speculator. This area may have been better utilised, for example, by increasing cache size.

Keppel [Kep96] shows that run-time value-specific optimisation performed by software can improve the performance on certain benchmarks. Keppel only considered a small region to apply his value-specific optimisations, constant propagation over a larger region would increase the benefit of the optimisation as in Dynamo (see section 3.2.2).

Owen [Owe00] presents a scheme for exploiting memory reference patterns by snapshotting memory activity within a hot block (identical to a group block). As the snapshot only reflects a short amount of program execution it is not fully accurate. Statistical analysis as to the accuracy of the snapshot can be used to drive dynamic optimisation which in certain circumstances are able to eliminate 12% of the instructions within a hot block. The snapshot results show that on average one third of all loads within a hot block load a constant value. The snapshot is also able to direct alias analysis.

### 4.3.8 Code duplication

Code duplication is a way of specialising code to a particular instance of its use using the compatibility mechanism presented in section 4.3.3. It is presented in RSJ<sup>+</sup>03.

## 4.4 Back-end

A TCODE back-end implements a set of functions for planting dynamic code routines in to the dynamic code buffer currently under construction. For example, a call to the `STWIRI` function of the back-end will plant code to store an immediate value at the destination address given by the addition of an immediate and register operand. In a similar manner the back-end provides successor functions that will plant code for Dynamite's branches. As part of successor and group block creation labels are generated and used by TCODE jump and branch instructions.

To plant efficient code it is not enough merely to have an appropriately rich back-end interface. There are a number of optimisations that can be performed during and after code generation as described in the following sections.

### 4.4.1 Instruction scheduling

A key to getting performance from modern pipelined and superscalar processors is scheduling instructions. Processor pipelines have bubbles inserted to ensure instructions are executed in order. As the bubbles take the place of instructions the clock cycles per instruction (CPI) increases. This degrades the performance of the processor compared to an optimally scheduled piece of code that has no

next:	U pipeline	V pipeline
addl \$4,%esi	addl \$4,%esi	Idle
movl -4(%esi),%eax	Idle (AGI)	Idle
shll \$2,%eax	movl -4(%esi),%eax	Idle (Read-after-write)
addl \$4,%edi	shll \$2,%eax	addl \$4,%edi
movl -4(%edi),%edx	Idle (AGI)	Idle
addl %eax,%edx	movl -4(%edi),%edx	Idle (Read-after-write)
addl \$4,%ebx	addl %eax,%edx	addl \$4,%ebx
movl -4(%ebx),%eax	Idle (AGI)	Idle
decl %ecx	movl -4(%ebx),%eax	decl %ecx
jnz next	jnz next2	

next:	U pipeline	V pipeline
movl (%esi),%eax	movl (%esi),%eax	movl (%edi),%edx
movl (%edi),%edx	shll \$2,%eax	addl \$4,%esi
shll \$2,%eax	addl %eax,%edx	addl \$4,%edi
addl \$4,%esi	movl (%ebx),%eax	addl \$4,%ebx
addl %eax,%edx	decl %ecx	jnz next
addl \$4,%edi		
movl (%ebx),%eax		
addl \$4,%ebx		
decl %ecx		
jnz next		

Figure 4.3: Instruction scheduling on an Intel Pentium processor

bubbles in the processor pipeline. Figure 4.3 shows an example of two pieces of IA-32 code and the 2 pipelines (U and V) of the Intel Pentium processor [CKK<sup>+</sup>95]. The first piece has a CPI of 1.0 clocks per instruction, the second piece has a CPI of 0.5 clocks per instruction (assuming cache hits for all memory accesses). The pieces of code are equivalent and both consist of 10 instructions. The second piece of code will execute in half the time it takes to execute the first piece of code so performance has been doubled of this loop. The pipeline bubbles in the first piece of code are caused by address generation interlocks (AGI), where a register changed in the preceding clock cycle is used as a base address in the current one. There are also bubbles in the pipeline due to registers not being available for reading until after an instruction has completed.

Scheduling has the ability to improve performance drastically and it does this by exploiting instruction level parallelism (ILP). The traversal of IR graphs to

improve instruction scheduling is presented in How00, OAHH03b, OAHH04.

#### 4.4.2 Idiom recognition

Replacing long sequences of code with shorter ones is known as idiom recognition. This is described in OAHH04.

#### 4.4.3 Code generation

The code generation of Dynamite is described in OAHH03b, OAHH04.

### 4.5 Summary

This chapter has introduced Dynamite the framework on which the Dynamite JVM is built. DBTs are significantly different to JVMs so there are certain design issues and opportunities unique to making a JVM from a DBT. This is described in the next chapter.

The converse to the 90/10 rule [HP96] is that 10% of execution time is spent in code that is infrequently executed such as initialisation routines. The time to translate code rises exponentially as the performance of that translated code increases (as shown figure 4.4<sup>1</sup>). As the translation cost will never be recouped for code that is executed once, it is better to just interpret. This factor is recognised by the HotSpot JVM (see section 2.10) that moves between three levels of execution and compiled/interpreted code performance.

---

<sup>1</sup>Figure 4.4 is drawn from communication with numerous JVM authors.

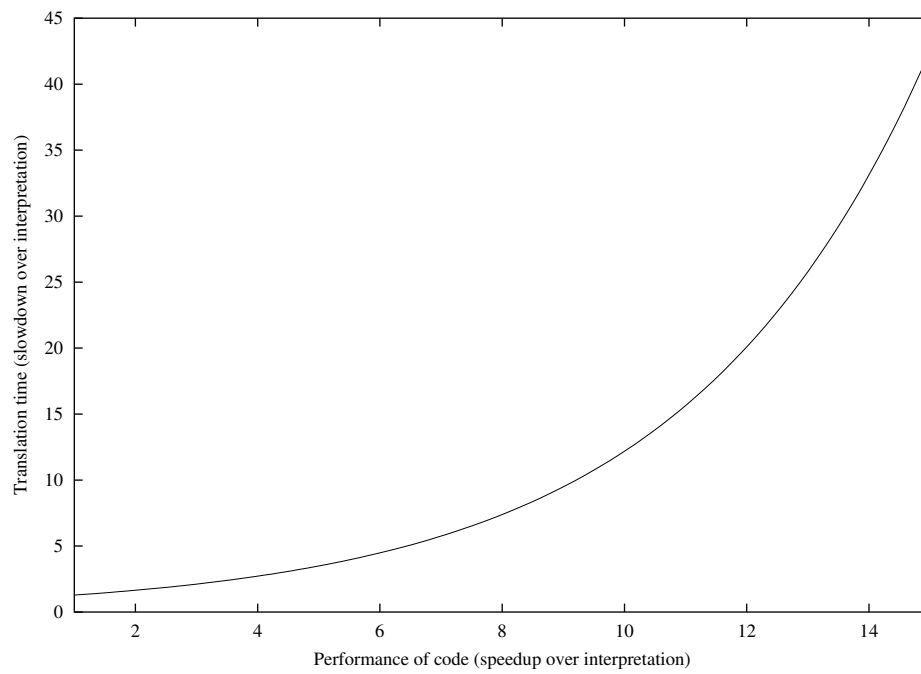


Figure 4.4: Estimated performance of code vs. translation time

# Chapter 5

## Dynamite JVM

The Dynamite JVM combines the Dynamite DBT with the execution of Java programs. Chapter 4 described the Dynamite DBT and chapter 2 introduced the Java Virtual Machine.

This chapter describes the basic components of the Dynamite JVM; the following chapter goes on to describe how inter-procedure optimisations are performed within the Dynamite JVM. Particular focus is placed on inter-procedural optimisation as it is key to the performance of object-oriented environments. Chapter 7 describes how the Dynamite JVM deals with a complication of inter-procedural optimisation, recursion. Chapter 8 considers the overall system performance of the Dynamite JVM.

### 5.1 Instruction decoding

Bytecodes within the JVM are uniquely identified by a PC value relative to the beginning of a method. The PC is not a visible register and is only used to index bytecode for branches. To uniquely identify basic blocks within the Dynamite DBT a subject address is required. The Dynamite JVM uses the address in memory the bytecode has been loaded into. This value is guaranteed to be unique, but if a method is ever unloaded the Dynamite JVM's kernel basic block cache for this subject address should be flushed.

The instruction decode of the Dynamite JVM classifies bytecodes into 3 types:

- Expression bytecodes: bytecodes that can be generated using Dynamite expression IR.

- Control-of-flow bytecodes: bytecodes that end a basic block and determine what the successors of this basic block are.
- Substitute bytecodes: bytecodes that cannot be implemented purely in IR and must trap out to a substitute routine.

Expression bytecodes are decoded using an expression stack described in section 5.1.1. Control-of-flow bytecodes are described in section 5.1.2. Substitute bytecodes are used to perform operations involving the virtual machine, such as object creation, or involving native method calls. The descriptions of substitute bytecodes appear throughout the following sections.

### 5.1.1 Expression stack

The use of an operand stack adds an overhead to Java bytecodes that is eliminated by folding several bytecodes together. This is performed in both hardware and software JVMs (see section 2.8 and 2.9). It is a requirement that operands on the bytecode operand stack are at the same depth every time a basic block is executed. This allows the stack to be mapped onto registers. Further optimisation of the operand stack can be performed with operands only needing to be on the operand stack for 7% of basic block boundaries [Kra98].

The Dynamite JVM allocates a stack frame for a translation to work within. The frame holds the local variables, frame data and operand stack. The frame data records information about the method such as the frame link pointer (a pointer to the previous stack frame on the call stack) and return address. The maximum sizes of the local variables and the stack are used to calculate the position of data within the stack frame. This is hidden in the implementation of the Dynamite JVM using programming language abstraction.

To eliminate the operand stack overhead, instead of reading and writing registers, bytecodes manipulate an expression stack built up from Expression IR. Figure 5.1 shows a translation made by the Dynamite JVM to IR. Figure 5.2 shows the contents of the expression stack at each PC value during the first part of the translation.

The expression stack is a stack of pointers to the DAGs of IR that define a particular stack location at any point of the translation. If a value needs to be read and there is no IR in the expression stack then the translator reads a value from the registers the stack maps onto.

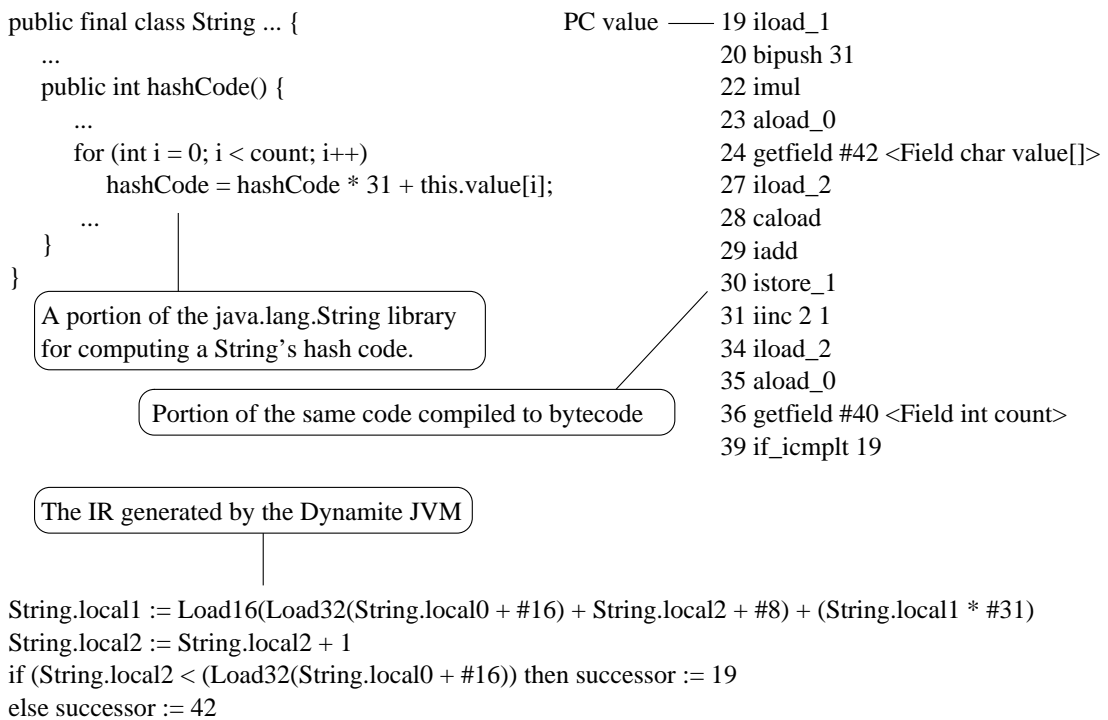


Figure 5.1: Example Dynamite JVM translation from bytecode to IR

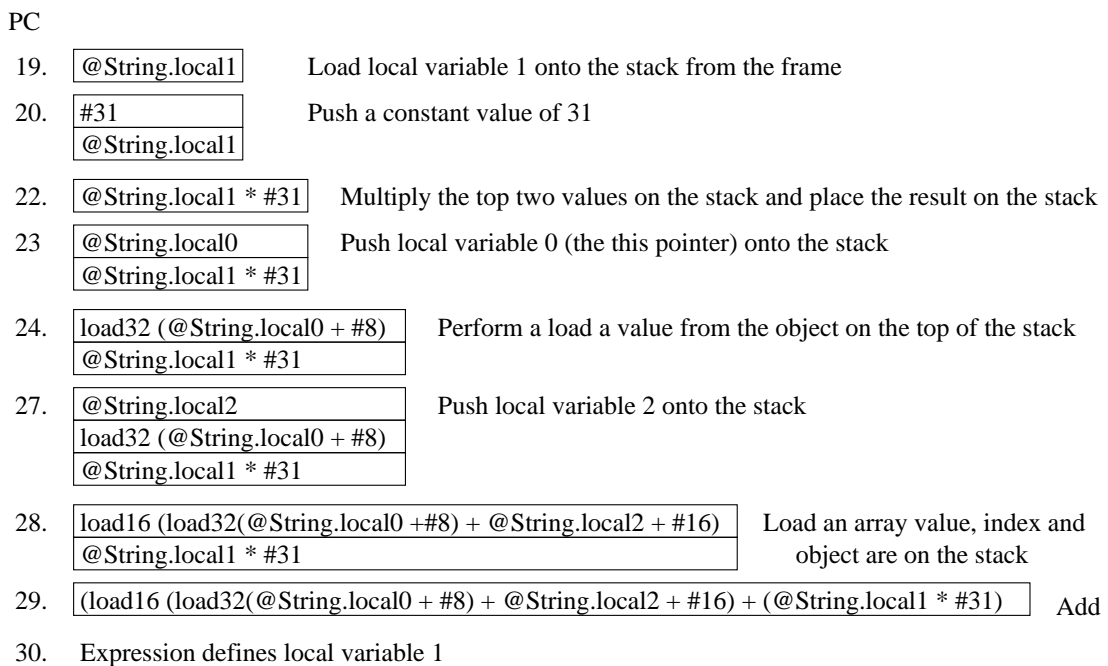


Figure 5.2: Expression stack during translation

The expression stack can ideally eliminate the use of stack registers from 93% of basic blocks. Meanwhile, the local variables are optimised, just as abstract registers are, by dead code elimination, constant propagation and expression sharing optimisations described in section 4.3.

### 5.1.2 Control-of-flow bytecodes

Java has 4 kinds of control-of-flow bytecode:

- method call: there are 4 kinds of method call in Java. `Invokestatic` and `invokespecial` call a fixed method whereas `invokevirtual` and `invokeinterface` are used to call a method associated with an object. Virtual methods are used to support inheritance in Java; interfaces are used as an approximation to multiple inheritance as described in chapter 2. `Invokespecial` differs from `invokestatic` as it passes an object with the method call. Method calls in bytecode differ from more conventional calls as not only is the call responsible for altering the stack frame and recording the return address, the method call is also responsible for passing parameters between frames. The top variables on the stack become the first local variables of the called method. To save copying parameters frames can be overlapped.

Chapter 6 describes the optimisation of method call bytecodes within the Dynamite JVM.

- branch: There are a number of different bytecodes for performing branches either unconditionally or conditionally. `Goto` is used to branch to a fixed PC location within the method. `If_icmp` and `if` have different variants for comparing integer values on the stack with each other or with zero. To perform comparisons of other primitive types such as doubles or floats, they must first be compared using a comparison instruction like `fcmplt` that yields an integer result that can then be used as an operand for a branch. `Ifnonnull`, `ifnull` and `if_acmp` are used like the integer branches except to compare object references.
- switches: `Tableswitch` and `lookupswitch` are used to perform more complicated integer branches. `Tableswitch` provides a lookup table of branch locations that is indexed by an operand on the top of the stack. The lookup

table is of a limited size and if the operand lies outside of the range of the table then a default location is branched to. `Lookupswitch` is similar to `tableswitch` except it provides pairs of value and branch location. If the value on the top of the stack matches a `lookupswitch` value then the corresponding branch location is branched to. If no value matches then again a default location provided by the instruction is branched to.

- finally: Finally clauses in Java are implemented using a pair of instructions: jump subroutine (`jsr`) and return (`ret`). `Jsr` pushes the current PC location onto the stack and branches to a location within the method. `Ret` takes a value for a local variable and branches to that location within the current method. The PC location can never be inspected by the virtual machine, it is just used in this pairing. Any bytecode that attempts to read the PC value will cause a JVM error to be raised during bytecode verification. The value is moved from the stack to the local variable by using the `astore` bytecode that has a special caveat that not only can it place object references from the stack into local variables, but values of type `returnAddress` too.

Method call within the Dynamite JVM is performed using a constant jump for `invokestatic` and `invokespecial`. As translation of the call means it will be executed, the location can be immediately resolved for the successor address. Section 5.3 describes how virtual calls are dispatched using a virtual method table (VMT).

Branches are implemented in the regular way except operands are fetched from the expression stack to avoid register manipulation. `Tableswitch` can be implemented using a computed branch successor. `Lookupswitch` requires a successor type which will compare multiple values and choose an appropriate branch location. This does not match any successor type currently supported by the IR. The addresses in both switch operations are big endian, the Dynamite JVM endian converts the tables and then alters the bytecode to either a `tableswitch_quick` or `lookupswitch_quick` bytecode, if the target machine is little endian. These `_quick` bytecodes are specific to the Dynamite JVM.

Both switches use IR that is not currently supported by the Dynamite JVM, they are therefore implemented as substitute calls. Switches are relatively infrequent bytecodes; they make up to 0.7% of a program's dynamic instruction

mix [RRJ99], so performance is not as key as with other bytecodes. In the long-run, if the successor types were not added to the IR then performance can be improved by replacing switches with special dummy bytecodes. These dummy bytecodes will branch to a piece of IR that will implement the switch statement and then branch to the correct target. Group block optimisation can remove the branch overhead introduced, but instruction translation will have been complicated.

`Jsr` is decoded and IR generated that places the address of the next bytecode pushed onto the stack. `Ret` reads the value from a local variable register and performs a computed jump to it.

## 5.2 Exceptions

Exceptions require the JVM to be able to generate a call-stack trace. The Dynamite JVM creates the call-stack trace by traversing the frame data sections of a method's call frame. In the frame data is a link register which holds the address of the previous method's frame data. The frame data also holds a method identifier so that the stack values can be interpreted. These data structures are discussed further in chapter 6.

The Dynamite JVM performs exception dispatch using stack unwinding as described in section 2.6. As with optimisations described by Lee et al. [LYK<sup>+</sup>00], exception handlers are translated when they are branched to by the exception mechanism. As exceptions are generally not on the main execution path then this reduces the amount of translated code.

A JVM has the ability to generate a large number of exceptions during loading, linking, resolving, verification and bytecode translation. The Dynamite JVM does not perform bytecode verification currently (this is typical of research virtual machines). Exceptions that occur in the other phases described above have their exception object generated and then thrown to the appropriate handler using stack unwinding.

Exceptions occur when a method is running for two reasons, an exception is explicitly thrown or a bytecode operation generates an exception. An explicit throw of an exception is generated by the `athrow` bytecode. This is implemented as a substitute routine by the Dynamite JVM. The substitute routine creates the stack trace and exception object, then throws the exception in the appropriate

method by stack unwinding.

JVMs also have to support a number of runtime exceptions that are not thrown explicitly within methods by the `athrow` bytecode. Null pointer exceptions occur when a non-existent object is referenced. The Dynamite JVM uses the UNIX segment violation signal to capture null pointer exceptions. The JVM uses an internal `JNULL` symbol that points to a page of memory that when accessed will cause the operating system signal to be generated. In certain cases this performance can be improved; for example, method dispatch on a null object. Incomplete support exists in the Dynamite JVM for speeding the capture of the null pointer exception caused this way by providing a dummy virtual method table (VMT) that branches into exception handling routines.

Arithmetic exceptions are generated by executing `idiv` or `irem` bytecodes and performing division by zero. The virtual machine generates an exception object as with the null pointer exception and then throws the exception.


To improve the robustness of arrays, JVMs have to catch array accesses that use an index that is out-of-bounds (less than zero or greater than the array size). Array accesses are very frequent within Java, hence the support of arrays as primitive types. The IR can generate exception handling code as shown in figure 5.3.

```

1  aload_1      // Get array reference from local variable 1
2  iconst_0     // Constant 0
3  iaload       // Load from array

stack0 := load (local1 + 0 + #8)
if ((0 < 0) || (0 > load (local1+#4))) successor := BoundsException
else successor := 4

```


  
simplified expression after  
constant propagation

(0 > load (local1+#4))

Figure 5.3: IR for an array out-of-bounds exception

The branch location `BoundsException` is a special area of memory known to the translator, which has a substitute call in it to perform the array index out-of-bounds exception.

The drawback to handling an array access in this way is it terminates a basic block early. This reduces the amount of potential optimisation that can be

performed by the expression stack and Dynamite JVM kernel. What is really required are bounded load and store operations in the IR. As out-of-bound exceptions are not generated by the benchmarks used in chapters 6 and 8, the Dynamite JVM ignores them by default in the current implementation. If required, a fall-back substitute call routine version of the JVM can be run that generates array index out-of-bounds exceptions.

### 5.3 Object layout

Object layout affects the performance of systems such as method dispatch and garbage collection. An object has to hold all the fields defined in the class file, fields inherited from superclasses, thread locks, an object's class and tables to dispatch virtual and interface method calls.

Using a handle allows greater freedom in garbage collection as objects can be moved around memory and only the handle needs to be altered to reflect this change [HGH96]. However, using a handle requires the translated code to perform two indirections to access a field. The first memory access reads the address of the object's data from the handle, the second read or write accesses the field. For performance reasons, the Dynamite JVM uses handleless objects as these only require one indirection to access a field.

Figure 5.4 shows the layout of an object in the Dynamite JVM.

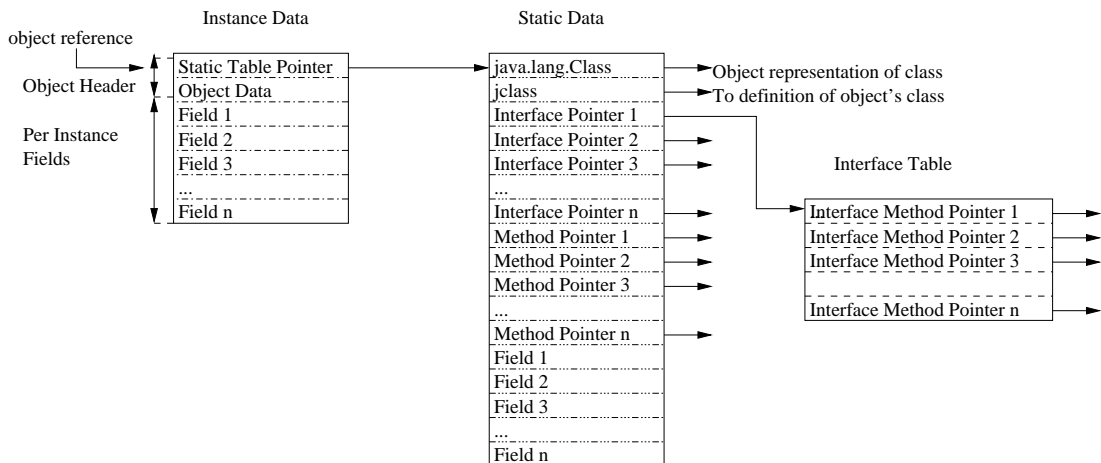


Figure 5.4: General layout of a Dynamite JVM object

Each object has a 2 word header that contains a pointer to the static data associated with it and a data field. Currently the object data field is used to hold

information on whether this object is an array and the type of the array. It is envisaged this field will be used also for synchronisation on objects and possibly garbage collection. The bits used to identify array information will be moved into the static data area. Dice [Dic01] describes a fast implementation of Java monitors using a single object data word.

Instance fields within the object vary in size from 1 bit to 8 bytes. For simplicity and alignment, the current Dynamite JVM uses a fixed 8 byte field size that can contain all the primitive types. This is space inefficient and can be optimised in future version of the Dynamite JVM. Arrays are implemented differently to objects, they use the first field as a size field to support the `arraylength` bytecode and to support index-out-of-bounds exception. The rest of the object contains the body of the array with elements adjacent to each other in memory.

The static data table holds a pointer to a `java.lang.Class` object that represents this class. This object is frequently used in library and native methods and is stored next to a `jclass` which is the virtual machines own internal representation of a class.

The static data table also holds a vector of interface table pointers. The size of this array is adequate for most applications. If a program requires more interfaces implemented then the tables can be grown. All translated code needs invalidating at this point. An alternative implementation negatively indexes interfaces from the start of the static data table. This allows the table to grow and for the translated code not to be invalidated, but it complicates memory allocation for the static data table. As growing the table is infrequently performed<sup>1</sup> the former scheme is currently used.

Interface table entries and the virtual method table contain pointers to the address associated with methods. Creating these tables requires the JVM to parse the class hierarchies of the classes and interfaces. Superclasses and superinterfaces appear at the beginning of each table with entries being overridden when a method of the same definition is found as the class hierarchy is traversed down to the implemented class or interface.

Static fields are recorded at the bottom of the static data table. Although this is not strictly required by the JVM specification [Sun95] it was considered good practice at the time of the design to keep all static data in one place.

---

<sup>1</sup>On all benchmarks run to date a table of size 64 has never needed growing.

## 5.4 Class loader

The Dynamite JVM's classloader provides coordination between dynamically executing code and the object model of the Dynamite JVM, as well as loading, linking, resolving and running the static initialisers of classes.

The Dynamite JVM has been designed to run using the Classpath [GNU02] class libraries. Classpath was chosen as it is not tailored to run on a pre-existing JVM and it provides native interface routes for both high-performance JVM integration (CNI) and portable JVM integration (JNI) (see section 2.4). Currently, the Dynamite JVM uses the JNI integration.

### 5.4.1 Boot strapping

Boot strapping is the initial phase of JVM execution. It is distinct from the main execution of a program to avoid race conditions introduced by cyclic dependencies (see section 2.2).

The first part of boot strapping is to create a thread for the application to run on. This is implemented using `java.lang.Thread` and adding this thread to the root thread group. A consequence of this is that the `java.lang.System` class is loaded and initialised. A call from the `java.lang.System` class initialiser to the JVM sets up system properties such as locale information. The final part of the bootstrap process is to create an array of strings to be passed to the main method that contains the command line parameters passed to the JVM.

### 5.4.2 New

New is implemented as a substitute call. The Dynamite JVM garbage collector is called to register an allocation so that it may free and run the finalizer on the object when it becomes unreachable. Currently, the Dynamite JVM supports exact garbage collection on basic block boundaries. Exact garbage collection is the ability of a JVM to differentiate objects from other forms of data; garbage collectors that cannot distinguish between types of data are referred to as conservative. A call is provided to generate a root set of nodes for the garbage collector to work with. From this set the garbage collector can determine what objects are reachable. However, the collect method of the garbage collector is not currently implemented. This also means that in the Dynamite JVM, finalize methods do

not get called.

After creating an object all the fields in that object must be initialised to their default value: 0 for numbers, false for booleans, null for object references. Calling the object's initialiser is handled by an `invokespecial` call in the bytecode instruction stream.

As well as creating new objects, there are variants of the new bytecode for creating arrays of varying dimensions. Multi-dimensional arrays are implemented as arrays of arrays, as defined by the Java specifications.

### 5.4.3 Checkcast and instanceof

`Checkcast` and `instanceof` bytecodes are used to check/determine whether an object on the operand stack is of the type given as a parameter to the bytecode. The complicated nature of these bytecodes requires the use of the JVM's internal representation of classes. `Checkcast` throws an exception if this check fails whereas `instanceof` replaces the item at the top of the stack with 0 or 1 dependent on whether the test is failed or passed. `Instanceof` also differs from `checkcast` as it pushes a 0 onto the stack if the object is null whereas this will cause an exception with the `checkcast` bytecode.

The operand at the top of the stack can be one of three classes of object:

- ordinary (non-array) classes: For a true result the object has to be the same class or a subclass of the one referenced in the instruction. If the instruction references an interface then the object has to implement that interface.
- interface: For a true result then the object has to implement the interface or a superinterface referenced by the instruction. If the instruction references a class then that class must be `java.lang.Object`.
- arrays: For a true result then the instruction must reference `java.lang.Object` or the interfaces implemented by an array (`java.lang.Cloneable` and `java.io.Serializable`). The instruction may also reference an array where a true result will be given if the class/type of the arrays are identical or the class of the object's array is a subtype of the instruction's array.

## 5.5 JNI

When a method static call bytecode (`invokestatic` and `invokespecial`) is translated the method to be called is known. By inspecting the flags associated with the method it is known whether this will be a native call and a substitute call can be planted in the target code.

The target native method has to be found from the list of native methods within the JVM. Extra native methods are loaded from libraries using the `java.lang.System.loadLibrary()` native method call. The Dynamite JVM uses the `dlopen` and `dlsym` Unix library calls [Ric00] to open and link dynamic libraries; the `libffi` foreign function interface library [Gre98] is used to ensure that parameters are properly packed for method calls.

For virtual and interface method calls the nature of the call, whether it is virtual or not, cannot be determined ahead of time. The Dynamite JVM instead points the VMT to a newly created dummy method containing one bytecode. The bytecode is called `fixup_iv_native` and is translated into a substitute call that performs the native call. The dummy method acts as a trampoline to the native code.

The JNI (see section 2.4) is a table of over 200 callback functions that is passed to all native methods as an environment argument. Most methods get passed an object parameter except static native methods that get passed the object representation of the class to which the static method belongs. The calls support accessing fields of classes and objects, creating objects, throwing exceptions, locking and unlocking objects, looking up classes and calling Java methods back in the JVM. This requires the Dynamite JVM to be re-entrant.

## 5.6 Threading

Threading in Java requires support for `monitorenter` and `monitorexit` bytecodes. These are used to implement synchronised regions of code within methods as shown in figure 5.5. Methods can also be flagged as synchronised and the associated object is locked/unlocked as the method is entered/exited, static methods use a lock associated with the object representation of their class. A lock appears in an object's header (shown in figure 5.4). Information contained in the lock is used for locking and unlocking the object. Java's threading protocol also requires

waiting and notifying to be performed on objects.

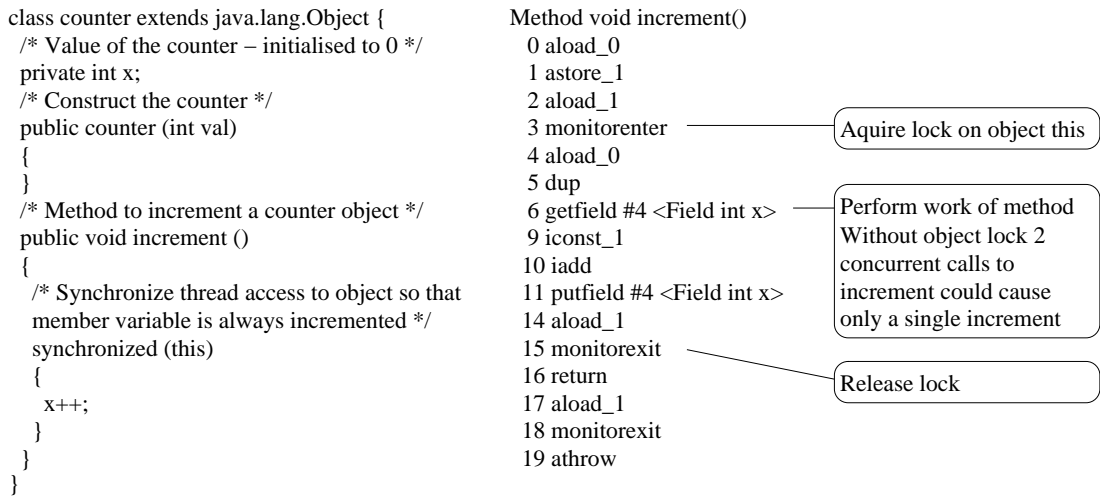


Figure 5.5: Bytecodes for a synchronised method

To implement threading a call stack is required per thread. This impacts on optimisations of the call stack as described in chapter 6. Possibly having multiple translations happening at the same time requires thread safety to be addressed within the Dynamite JVM front-end, kernel and back-end. A key place of contention is the basic block cache that would need locks placing on it to avoid invalidation by events such as the interface table overflowing (see section 5.3). As more of the subject program is translated, more time will be spent in dynamic code and contention on the basic block cache will be reduced.

The Dynamite JVM does not currently implement threads; the exact mechanism to be used would depend on the low-level implementation of threads available on the target machine. Support for parallel garbage collection and dead-lock detection can be implemented at the same time. Tuning would be required on the Dynamite JVM to avoid long waits on contended objects.

## 5.7 Related work

This chapter has introduced the Dynamite JVM. Noticeable omissions from the current JVM, that stop it from being certified as 100% Java compatible, are a garbage collector, support for multiple threads, and array bound exceptions. This chapter has discussed their implementation but the implementation is a matter for future work.

Taking an overview of the Dynamite JVM, it is interesting to compare it to other state-of-the-art JVMs that exist at the moment. The Dynamite JVM has multiple levels of execution. Initially it executes a basic block at a time until an execution threshold is reached. At this point a group block is formed, and later group blocks can also be formed incorporating a particular basic block. In contrast HotSpot has three distinctly separate phases of execution, interpretation, JIT (quick) compilation and an optimising compiler. Not having an interpreter in the Dynamite JVM slows it for methods that are only executed once as the translated overhead is never repaid by the faster code. The heuristic to determine when to compile is non-trivial though. Compilation that is based upon methods can find itself trapped in a particular method it is interpreting, that method needs to be exited for a translated version of this method to be used. It is for this reason HotSpot uses in-stack replacement to switch between its modes of execution.

The Dynamite JVM has the basic blocks trace scheduled within a group block. JVMs such as HotSpot and Jalapeno also trace schedule code but this is restricted to code within a method or inlined method. The heuristic of which method to inline is often simplistic, for example inlining of final and leaf methods. As the Dynamite JVM works with only enough knowledge of a method to support instruction semantics, it is not restricted by method boundaries when optimising; scheduling and optimisation is purely driven by execution statistics.

The Dynamite JVM has the potential to be a 100% compatible JVM using dynamic compilation and a mechanism for trace scheduling that is different from existing JVMs. To get the benefit of the freedom to trace schedule code the JVM must eliminate method boundaries fully and allow the code generation algorithms to view multiple call stack frames at the same time. This is the effect achieved by method inlining; inter-procedure optimisation within the Dynamite JVM is described in chapter 6.

To further improve the performance of the Dynamite JVM there are numerous sources of optimisation. Null method recognition is the ability of a JVM to realise that a method performs no operations. For example the initialiser of `java.lang.Object` is frequently called and all it does is return. By flagging frequently called methods such as this as null the JVM can avoid calling them when translating code. Methods that perform no operations and call a null method can themselves be marked as null, this is very common for Java constructors. This optimisation is only applicable to static calls.

The optimisation of exceptions can be performed as discussed in section 2.6. A key optimisation for exceptions is only generating stack traces when they are needed. This requires a detailed understanding of stack traces by the JVM and for it to inspect exception handlers to see if the stack trace is used.

Scientific applications make a lot of use of arrays and programming languages such as Fortran optimise them heavily (for example, reordering dimensions to improve cache performance). Java adds an extra indirection per dimension for multi-dimension array accesses, when compared to languages such as Fortran (as shown in figure 5.6).

```
A[i,j] = load(A+(i*column width)+j) in Fortran
A[i][j] = load(load(A+i)+j) in Java
```

Figure 5.6: Java array accesses compared to those of Fortran

As Java uses standard calls and standard mathematical libraries it can be reversed engineered, if necessary, and then optimised by the JVM. An example optimisation is the replacement of calls to `java.lang.Math.abs()` with translated code that calculates the absolute value of an integer. The same effect can be achieved at greater translation expense with method inlining.

Finally, optional features not required by a 100% Java compatible JVM can be added to the Dynamite JVM. For example, debug support can be added to a JVM through support for JVMDI [Sun01], the JVM debug interface/architecture.

## 5.8 Summary

This chapter has described core aspects of the Dynamite JVM. The Dynamite JVM has the potential to be a 100% Java compatible JVM and distinguishes itself from other JVMs in the way basic blocks are trace scheduled. The following chapter describes the implementation of inter-procedural optimisations within the Dynamite JVM. The chapter measures and analyses this cost of this optimisation, and the potential benefits it brings. Chapter 7 further refines and measures the inter-procedural optimisation mechanism. Chapter 8 analyses the remaining aspects of performance to the Dynamite JVM.

# Chapter 6

## Inter-Procedure Optimisation

In chapter 1 the problem of method call optimisation was introduced. Object-oriented programs tend to have short methods and small basic block sizes, this limits optimisations such as dead code elimination, constant propagation and instructions scheduling. To increase method and basic block sizes, method inlining is performed (see section 2.9.1). This is appropriate for JVMs that are oriented around the compilation of methods. The Dynamite JVM is different and is instead oriented around the compilation of basic blocks. This chapter describes two techniques allowing an approximation of method inlining that can be performed within the Dynamite JVM. These are evaluated and a final technique chosen.

The Dynamite JVM avoids translating code that does not contribute toward improving performance by using basic blocks as a compilation unit instead of methods. The techniques introduced here allow method inlining that is not based on assumptions about the program behaviour (as described in section 2.9.1) but, instead, on the execution threshold of basic blocks.

### 6.1 The method inlining problem

Method inlining places the body of one method inside the other. Figure 6.1 shows a high-level example of the inlining of a function `readChar` inside a function `readString` to create the function `readString_readChar`.

In figure 6.1 the function `readString_readChar` has both the stack frames of `readString` and `readChar` visible. By not pushing parameters onto the stack and popping them off again, the parameters are available to the compiler to optimise

```

class example {
    public char buffer[];
    public int ptr = 0;
    example() {
        buffer = new char[1024];
    }
    public String readString()
    {
        char buf[] = new char[26];
        int len=0;

        while(len < 26){
            buf[len] = readChar();
            if ((buf[len] == '\u0027')||(buf[len] == '\n')){
                break;
            }
            len++;
        }
        return new String(buf, 0, len);
    }
    public char readChar()
    {
        char result;
        if (ptr == 1024){
            result = '\n';
        }
        else {
            result = buffer[ptr++];
        }
        return result;
    }
    public String readString_readChar()
    {
        char buf[] = new char[26];
        int len=0;
        char readChar_result;

        while(len < 26){
            if (ptr == 1024){
                readChar_result = '\n';
            }
            else {
                readChar_result = buffer[ptr++];
            }
            buf[len] = readChar_result;
            if ((buf[len] == '\u0027')||(buf[len] == '\n')){
                break;
            }
            len++;
        }
        return new String(buf, 0, len);
    }
}

```

Figure 6.1: Method inlining

across the method boundary.

This chapter describes optimisations to be performed by the Dynamite JVM that expose frames over method boundaries whilst keeping the correct program execution semantics.

## 6.2 Sliding register-window scheme

Microprocessors such as the SPARC [SPA92] and IA-64 [Int01] map the call stack to registers and then slide a window up and down these registers as functions are called and returned. As indirect addressing is allowed within programming languages such as C (most commonly for local arrays), certain parameters are passed through memory to obtain the correct program behaviour. If the pool of registers is exhausted then an exception is thrown and the operating systems swaps registers onto the stack in memory.

The Dynamite JVM's IR does not provide any support for indirect addressing of registers. However, the compatibility function (see section 4.3.3) can be used to retranslate code, so the same basic block can have two translations using different abstract registers. This allows an approximation of register-windows at the expense of retranslation. Figure 6.2 shows an example translation using this scheme.

The group block mechanism can optimise over method boundaries in this scheme as multiple method frames are visible during the recompilation. Retranslation within this scheme is particularly expensive for recursive methods which require one translation per recursive invocation. To avoid paying this penalty for recursive methods, recursion detection (as described in chapter 7) can be used.

The sliding register-window scheme needs retranslation to occur when stack depths vary on a method call. To reduce this penalty it would be possible to try and align frames further and thus reduce the amount retranslation. For example, if all frames were the same size then a retranslation would only be necessary if the call depth varied rather than the call stack size varying.

## 6.3 Fixed register-window scheme

A fixed register-window scheme is one where each method is allocated a permanent set of abstract registers to use. Retranslation of a method is avoided at the

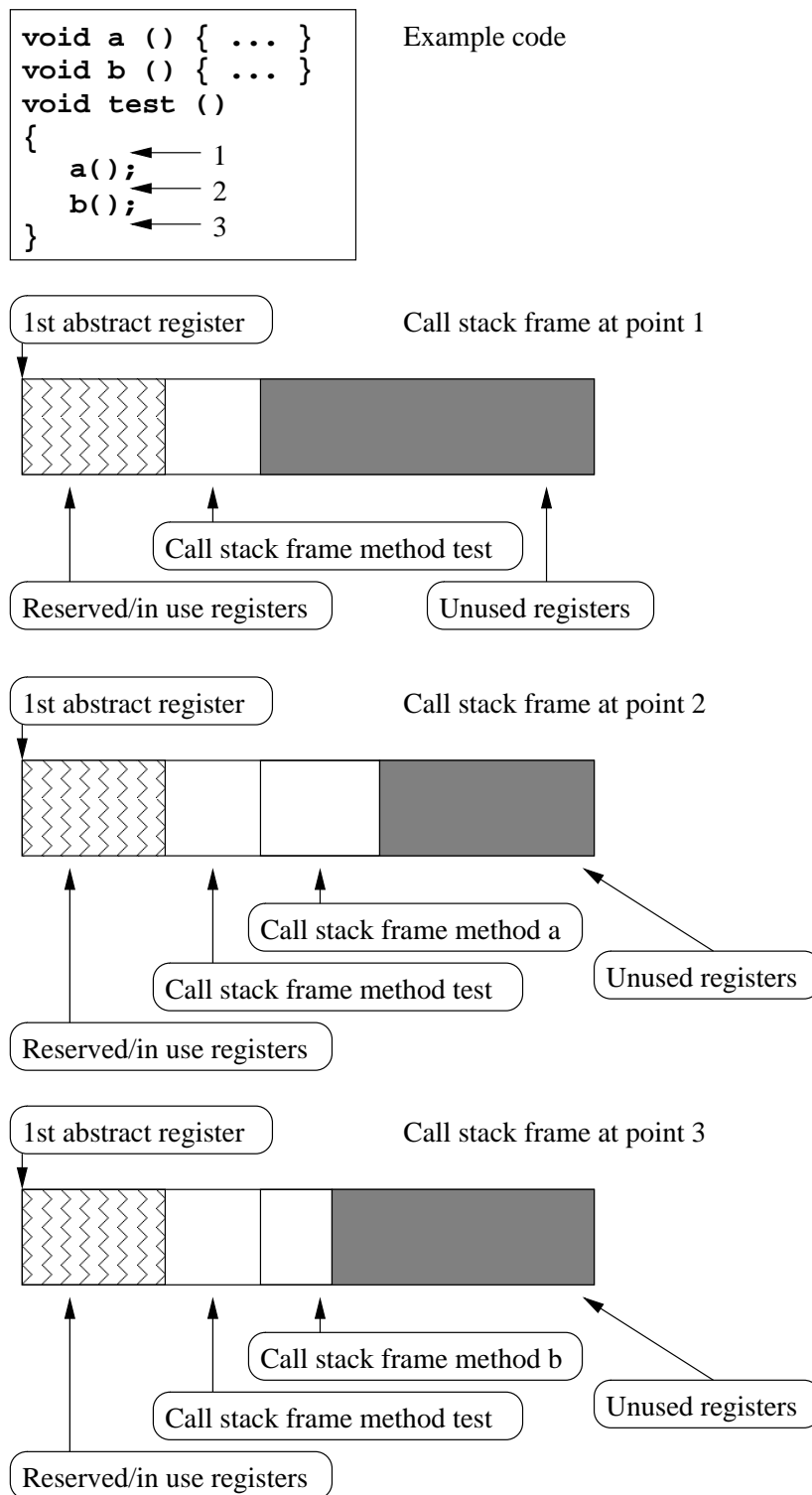


Figure 6.2: Sliding register-window example

cost of more expensive method calls (parameters can't be passed by overlapping stack frames). Figure 6.3 shows an example.

Static method call (`invokestatic` and `invokespecial` bytecodes) requires the calling method to place the frame data into the called method. Parameters are copied from the calling method and placed in the called method's local variables within its stack frame. As the parameters are on the expression stack, unnecessary register copying is avoided except in the case when the basic block prior to the calling basic block leaves something on the stack. This accounts for approximately 7% of basic blocks [Kra98]. However, it is not necessarily the case the parameters on the stack need passing to the called method. Group block optimisations can also dead code eliminate these values on the stack if they are subsequently overwritten.

Virtual method call (`invokevirtual` and `invokeinterface` bytecodes) requires a fix-up bytecode (`fixup_iv`) to be at the beginning of each method. This bytecode has to be 4 bytes long to ensure that `tableswitch` and `lookupswitch` bytecodes are still aligned. The fix-up bytecode is responsible for creating the frame data and copying the parameters from the calling method into the called method. The `fixup_iv` bytecode is translated by static `invokespecial` calls, but it performs no operation.

To determine the calling method the called method inspects the previous basic block pointer that is passed to the translation of the fix-up bytecode's basic block. The current version of the Dynamite JVM inspects the `this` object reference and finds the method from the virtual method table. Equally the method can be found by inspecting the subject address of the called method and looking it up in a table containing all loaded methods.

Parameter copying from the stack of the calling method to the locals of the called method is unavoidable for virtual calls with this scheme, however, group block optimisations should be able to eliminate them. An extra translation overhead is introduced as the basic block containing the fix-up bytecode is specialised to the call site and must therefore fail the compatibility test (see section 4.3.3) for calls from different call sites.

Bytecodes that return a value to the calling method (`ireturn`, `lreturn`, `areturn`, `freturn` and `dreturn`) leave the value on the top of their stack. The calling method checks the return instruction to see if a value should have been returned and places it on the expression stack at the start of translation. This

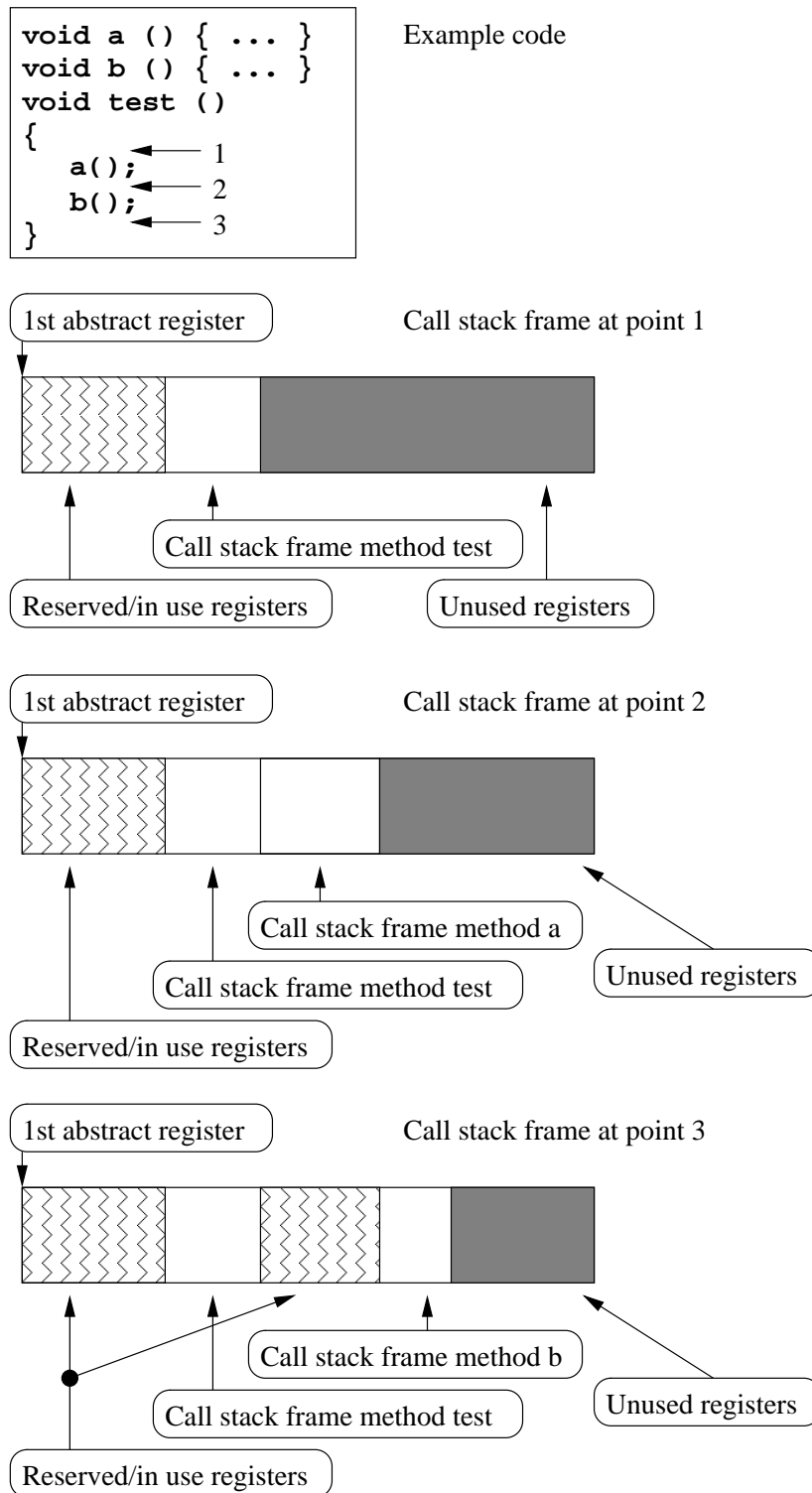


Figure 6.3: Fixed register-window example

specialises the basic block to the method that returns to it, and to support this, the basic block compatibility test is once again used.

To minimise the number of registers required, it is possible for leaf methods to share registers, as by definition they can not enter another method. This requires a method to be parsed before being executed, and could be implemented in a bytecode verifier that annotates methods for the JVM. In the current scheme, all translations are invalidated when the register pool is exhausted. This starts the process of translation again, with all the registers unassigned to methods. On the benchmarks presented in section 6.4.1, it was found a register pool of 8192 registers was never exhausted.

## 6.4 Choice of register-window scheme

The performance of the group block optimisations is similar for both schemes. The fixed register-window scheme uses more registers and relies on group block optimisations to remove parameter passing overheads for virtual calls. The sliding register-window scheme requires more re-translation than the fixed register-window scheme.

In this section we gather metrics using the Dynamite JVM to help with the choice of scheme to use. First of all we introduce a set of benchmarks that will be used to measure performance.

### 6.4.1 Introduction of benchmarks

As described in section 5.7 the Dynamite JVM is not 100% compatible with all Java programs. Due to limited resources not all benchmarks that could potentially be made to run were able to. This was largely down to bugs in the translation and class library. The chosen set of benchmarks are still a rigorous test of all parts of the JVM. They are described below:

- **helloworld**: A Java program that prints the string “Hello World!” to screen. Most of the executed bytecodes occur due to bootstrapping of the class library.
- **jmpeg2dec**: A Java program that decompresses several frames of an mpeg2 graphics file [EM01]. This benchmark was chosen due to the availability

of the source code and it reflects a realistic workstation workload. Correct execution of the program can be checked by viewing the decoded frames.

- `jpeg2enc`: A Java program that compresses several frames of pictures into an mpeg2 movie file employing numerous compression algorithms [EM01]. Again, correct execution of this benchmark can be verified. The benchmark is of the type of workload that would be run on a server.
- `_201_compress`: The compress benchmark from the SpecJVM 98 [SPE98] benchmark suite. This program compresses one of three possible sets of sample data. The benchmark harness verifies the correct execution of the program.
- `_202_jess`: The jess benchmark from the SpecJVM [SPE98] benchmark suite. This program is the Java Experts System Shell. The program itself reads in several descriptions and then uses them to generate results to questions. Two sets of input are considered for this benchmark and again the validity of the benchmark is checked by the benchmark harness.

This gives a total of five benchmarks, with `_201_compress` and `_202_compress` being run with medium sized data sets.

### 6.4.2 Number of fix-up blocks for fixed register-window scheme

Fix-up blocks are an overhead for the fixed register-window scheme. To gather statistics the Dynamite JVM was instrumented to record the number of `invokestatic`, `invokespecial`, `invokevirtual`, `invokeinterface` and `fixup_iv` bytecodes translated and executed by the JVM operating in this mode.

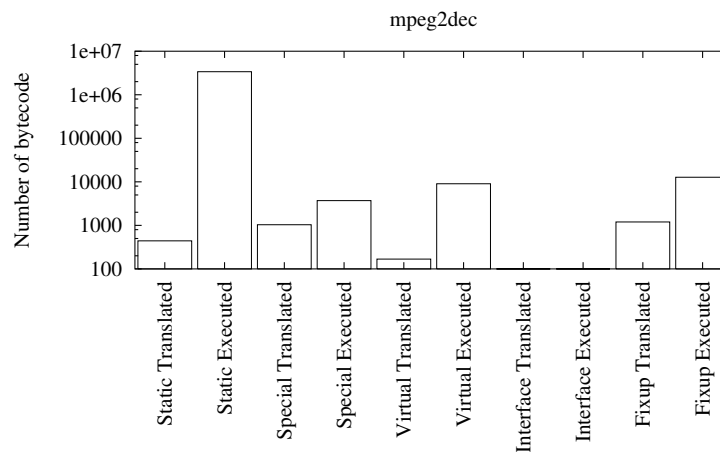
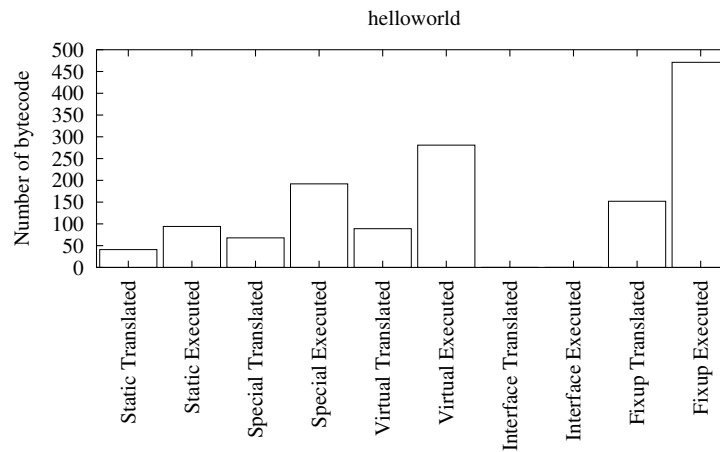
Table 6.1 shows the number of bytecodes translated and executed by each benchmark using the fixed register window scheme.

The number of bytecodes executed is shown in figure 6.4 (the bytecode names have been abbreviated).

Figure 6.4 shows some features that would not be expected purely from the fixed register-window scheme. The number of `invokespecial`, `invokevirtual` and `invokeinterface` bytecodes translated by each benchmark is greater than the number of `fixup_iv` bytecodes translated. It would be expected that the

Benchmark	Translated Bytecodes	Executed Bytecodes
helloworld	4,398	21,877
mpeg2dec	27,152	604,548,289
mpeg2enc	33,437	14,693,689,649
_201_compress	19,570	1,212,483,527
_202_jess	33,907	148,612,987

Table 6.1: Benchmark translation and execution bytecode counts



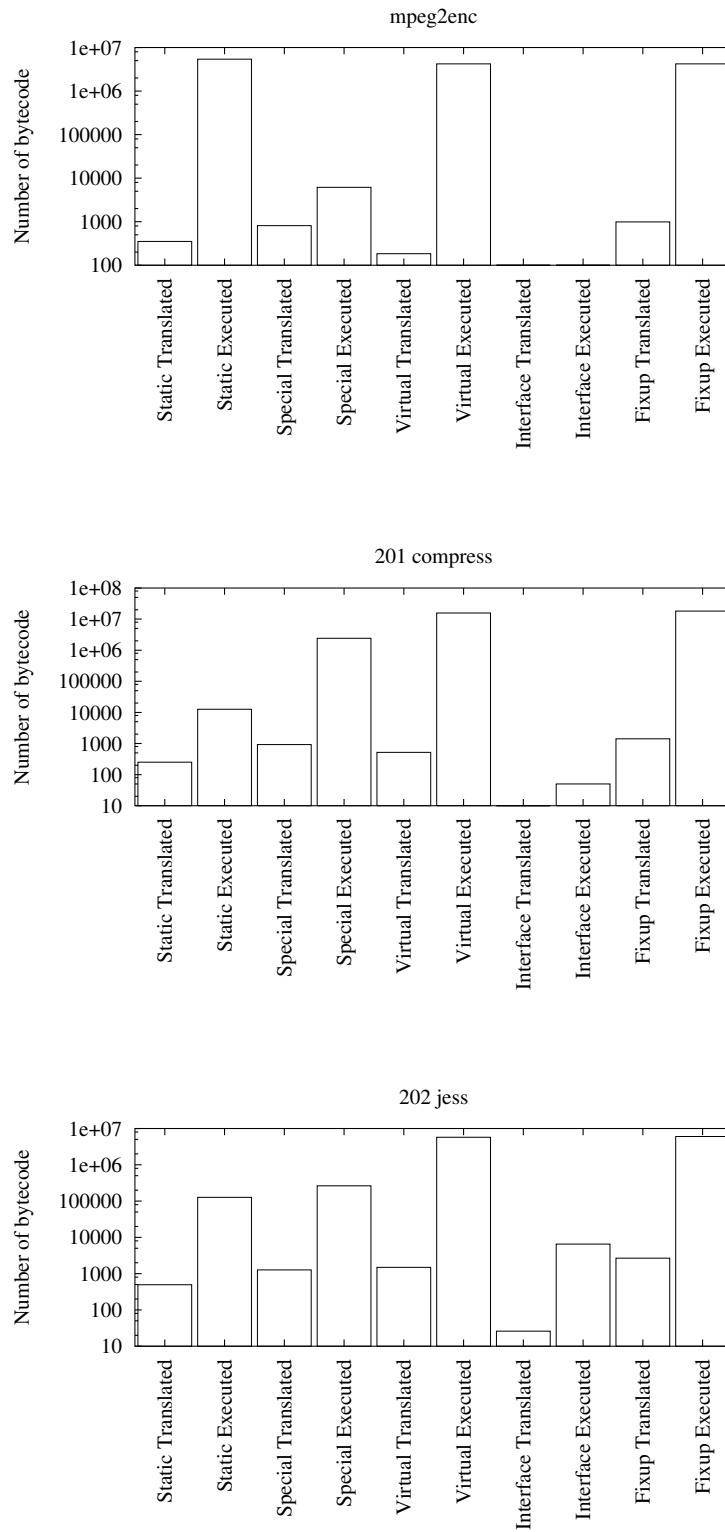


Figure 6.4: Fix-up overhead for fixed register-window scheme

helloworld	Translated	Executed	mpeg2dec	Translated	Executed
invokestatic	0.932%	0.430%	invokestatic	1.624%	0.560%
invokespecial	1.546%	0.878%	invokespecial	3.819%	0.000%
invokevirtual	2.024%	1.284%	invokevirtual	0.619%	0.001%
invokeinterface	0.000%	0.000%	invokeinterface	0.000%	0.000%
fixup_iv	3.456%	2.153%	fixup_iv	4.416%	0.002%

mpeg2enc	Translated	Executed	_201_compress	Translated	Executed
invokestatic	1.053%	0.037%	invokestatic	1.288%	0.001%
invokespecial	2.428%	0.000%	invokespecial	4.711%	0.200%
invokevirtual	0.550%	0.029%	invokevirtual	2.657%	1.301%
invokeinterface	0.000%	0.000%	invokeinterface	0.051%	0.000%
fixup_iv	2.955%	0.029%	fixup_iv	7.215%	1.501%

_202_jess	Translated	Executed
invokestatic	1.463%	0.085%
invokespecial	3.728%	0.177%
invokevirtual	4.403%	3.857%
invokeinterface	0.077%	0.004%
fixup_iv	7.895%	4.038%

Table 6.2: Percentage of method call bytecodes translated and executed in fixed register-window scheme

number of `fixup_iv` bytecodes would be greater, as single `invokevirtual` and `invokeinterface` bytecodes can go to multiple locations and thereby cause the translation of specialised fix-up blocks (containing `fixup_iv` bytecodes). The reason for the extra method call bytecode translations is due to invalidation of basic blocks caused by recursion, this is explained in chapter 7. A small number of the method calls also go to native methods that don't use `fixup_iv` bytecodes (see section 5.5).

Table 6.2 shows the translation and execution statistics for the measured bytecodes on each benchmark. Appendix B.1 contains the raw data used to create these results.

The `fixup_iv` bytecode accounts for less than 2% of the overall dynamic instruction mix. When `invokespecial` calls are factored out (these do not require extra parameter copying as they are static calls) the `fixup_iv` accounts for less than 1.3% of the dynamic instruction mix.

Table 6.3 shows the parameter copying overhead per virtual call executed by the benchmark. The average number of parameters passed by `invokevirtual` and `invokeinterface` bytecodes shown in table 6.3 is greater than 1 as all virtual

calls must pass a `this` pointer.

Benchmark	Copying overhead (average number of parameters copied)
helloworld	1.947
mpeg2dec	2.675
mpeg2enc	1.019
_201_compress	1.923
_202_jess	1.975

Table 6.3: Fixed register-window copying overhead per virtual call

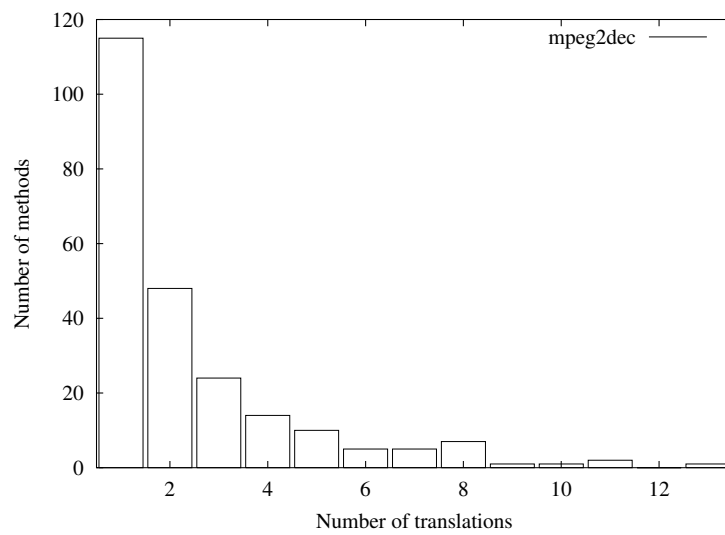
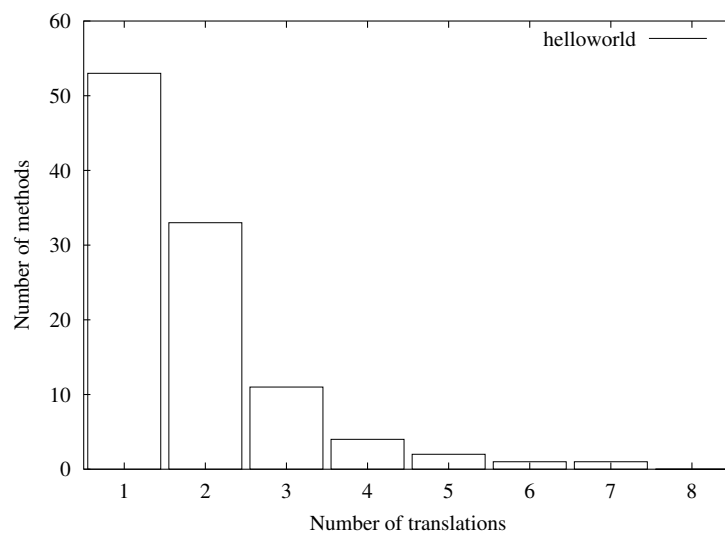
### 6.4.3 Number of retranslations for sliding register-window scheme

The Dynamite JVM was instrumented to record what depths of call stack a method is executed at. For each of these depths, with the sliding register-window scheme, a translation is required. In fact more translations are actually required as the call stack is not guaranteed to be the same depth for a particular depth of call (a Java stack frame isn't a fixed size). A fixed frame size can be used but some method calls will need registers containing parameters moving to appear in the correct location.

Figure 6.5 shows the number of translations required for each method using a sliding register-window scheme with one translation required per stack depth. Appendix B.2 shows these results in a table format.

Benchmark	Translated SRW	Translated non-SRW	Ratio	Bytecodes executed
helloworld	191	105	1.819	21,877
mpeg2dec	564	233	2.421	604,548,289
mpeg2enc	477	247	1.931	14,695,311,595
_201_compress	1086	386	2.813	1,212,483,557
_202_jess	2618	553	5.103	148,613,239

Table 6.4: Overall sliding register-window statistics



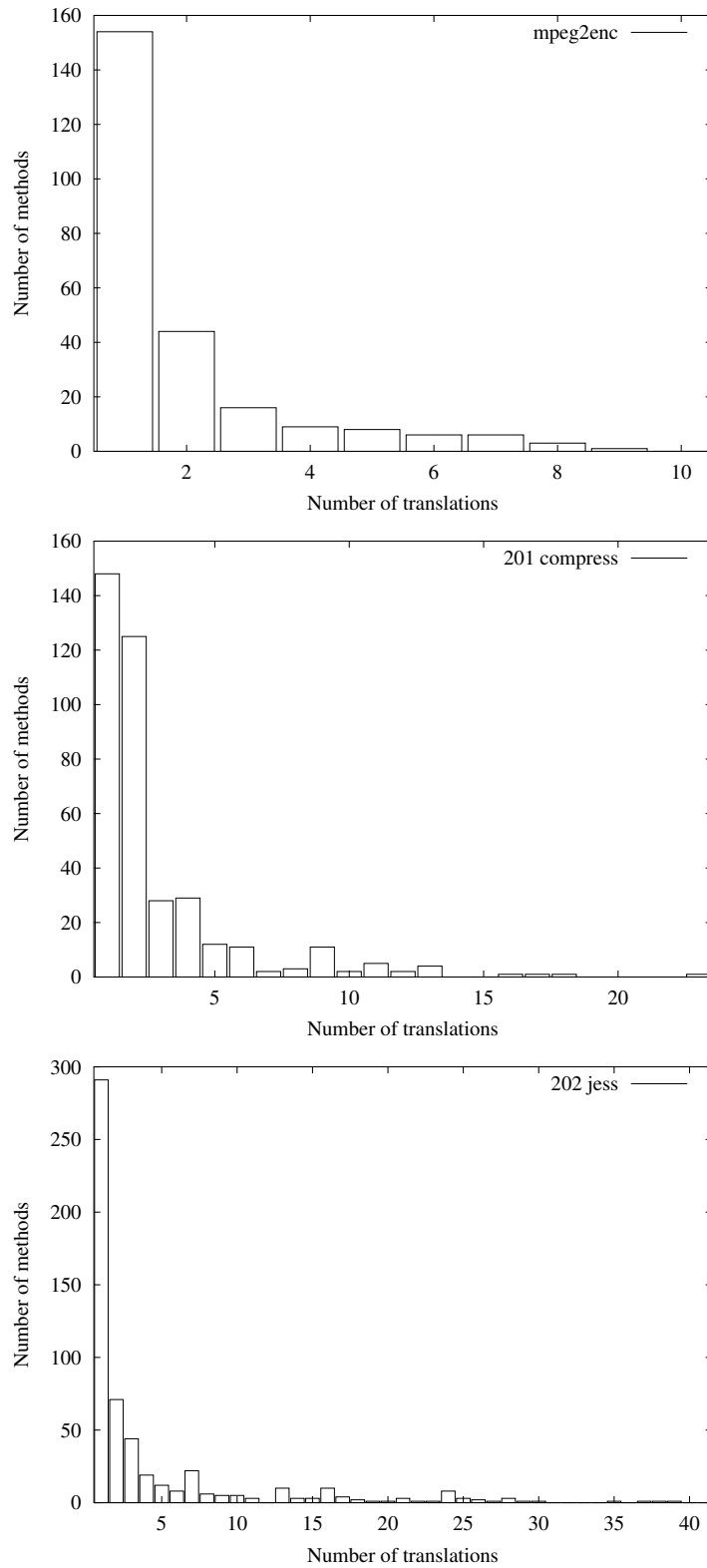


Figure 6.5: Number of translations required for the sliding register-window scheme

Table 6.4 shows the number of translations performed per method both with and without the sliding register window (SRW) scheme. The ratio gives an average number of times a method will be translated with the sliding register-window scheme compared to a scheme without a sliding register-window. The final column shows the number of bytecodes executed by a benchmark.

The number of bytecodes executed vary from table 6.1 as slightly different execution paths are taken within benchmarks. Typically this is because of time and pseudo-random number generator functions.

The results show that methods will be translated over 1.819 times more using a sliding register-window scheme compared to a scheme without a sliding register-window. The ratio shows no clear correlation to the total number of executed bytecodes. The results show that close to twice as much translation has to be performed when using the sliding register-window scheme. This will greatly increase the translation cost, with some benchmarks (`_201_compress` and `_202_jess` benchmarks in particular) performing worse than others.

#### 6.4.4 Coverage of fixed register scheme

The Dynamite JVM uses a greedy register allocator (as described in section 4.4.3) to allocate registers to instructions as they are translated. As the Dynamite JVM eliminates stack register usage for approximately 93% of basic blocks (see section 5.1.1), the main remaining register usage is for local variables. Frame data is also stored in abstract registers, but these are only accessed on method boundaries.

By instrumenting the Dynamite JVM and Kaffe JVM [Wil02] execution statistics were gathered on methods and the number of registers they require. The number of registers required was estimated to be the number of local variables. Not all the frame data and local variables will be used in a basic block, but the ability to reuse or not to allocate for these registers, by the register manager, is ignored.

In previous work [RRS99] it was shown that a large percentage of the total execution time of a program could fit into a group block without requiring the Dynamite JVM to perform register spill and fill operations. It was found that with 5 target machine registers 22% of the javac [Sun99a] program's dynamically executed code could fit into a group block without requiring spill or fill operations. With 26 registers, 46% of the dynamically executed code can fit into a group

block without requiring a spill or fill operation. By changing the heuristic to favour methods that contribute to runtime and use few local variables (results were sorted based on their percentage of execution time per local variable), it was found that with 8 registers 30% of the total execution would fit in a group block and with 25 registers 54% would fit without requiring register spills or fills.

The study was repeated using the Dynamite JVM on the set of benchmarks described in section 6.4.1, the results are shown in figure 6.6.

Figure 6.6 shows that with 100 registers (local variables) over 80% of a program's dynamic execution can fit into a group block without register spills or fills. The amount that fits depends on the benchmark, with `mpeg2dec` and `mpeg2enc` using more local variables in their hot regions than `helloworld`, `_201_compress` and `_202_jess`.

### 6.4.5 Conclusion

Two schemes have been presented for allowing the Dynamite JVM's group block mechanism to optimise registers over method boundaries.

- sliding register-window scheme: The sliding register-window scheme requires a translation of a method's basic blocks for each particular depth it is executed at. The number of translations required varies between benchmarks, but in the chosen set of benchmarks it is conservatively shown to be over 1.819 times per method.
- fixed register-window scheme: This scheme requires a specialised basic block that handles parameter passing for virtual calls. The average number of parameters passed in the chosen set of benchmarks was typically less than two. The number of fix-up bytecodes required was found to be as much as 4.3% of the translated bytecodes and 1.8% of the executed bytecodes.

As the overhead of the fixed register-window scheme can be eliminated by group block formation, it is believed this scheme is better than the sliding register-window scheme that must always incur extra translation penalties.

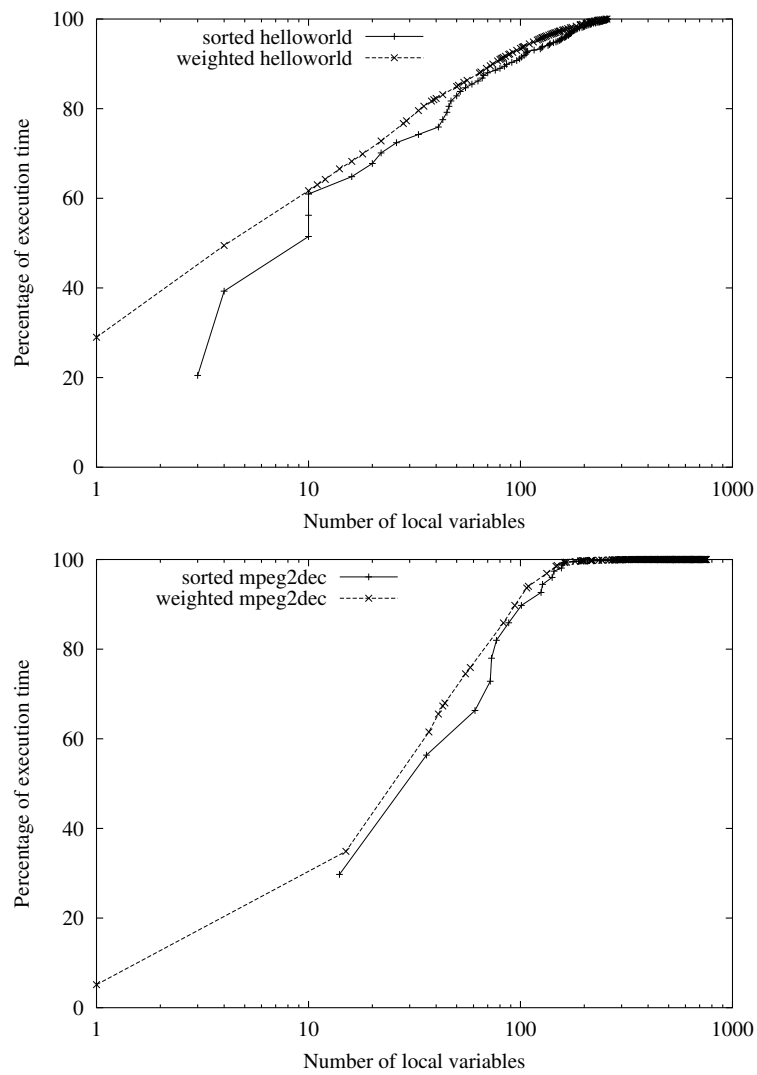
In a group block the method calls to fixed (static) locations are removed and the basic blocks trace scheduled to reduce the cost of moving between them. Virtual method calls and method returns require a computed jump to calculate their destination. The computed jump for the method return can be removed

through constant propagation of the return register value. The computed jump for the virtual method call can be removed by value specific optimisation (see section 4.3.7). Currently within a group block, for computed jumps, the inline cache is used to schedule the basic blocks to appear after the method call and return. A test is performed to ensure the following basic block is the intended block. Conventional method inlining wouldn't require these tests, so for the Dynamite JVM's optimisations to be truly comparable to method inlining value specific optimisation and constant propagation are required within group blocks.

## 6.5 Summary

This chapter has presented two schemes for allowing the Dynamite JVM to perform a method inlining style optimisation. A novel technique has been chosen, based on benchmark results, that allows the Dynamite JVM translator to optimise across method boundaries by mapping a method's stack frame to a fixed set of registers. This incurs a parameter passing penalty that can be eliminated by group block formation.

As each method has an associated set of registers, recursive method calls will require the reuse of these registers. Chapter 7 describes how the Dynamite JVM deals with this problem. The techniques described are applicable to both the sliding register-window and fixed register-window schemes.



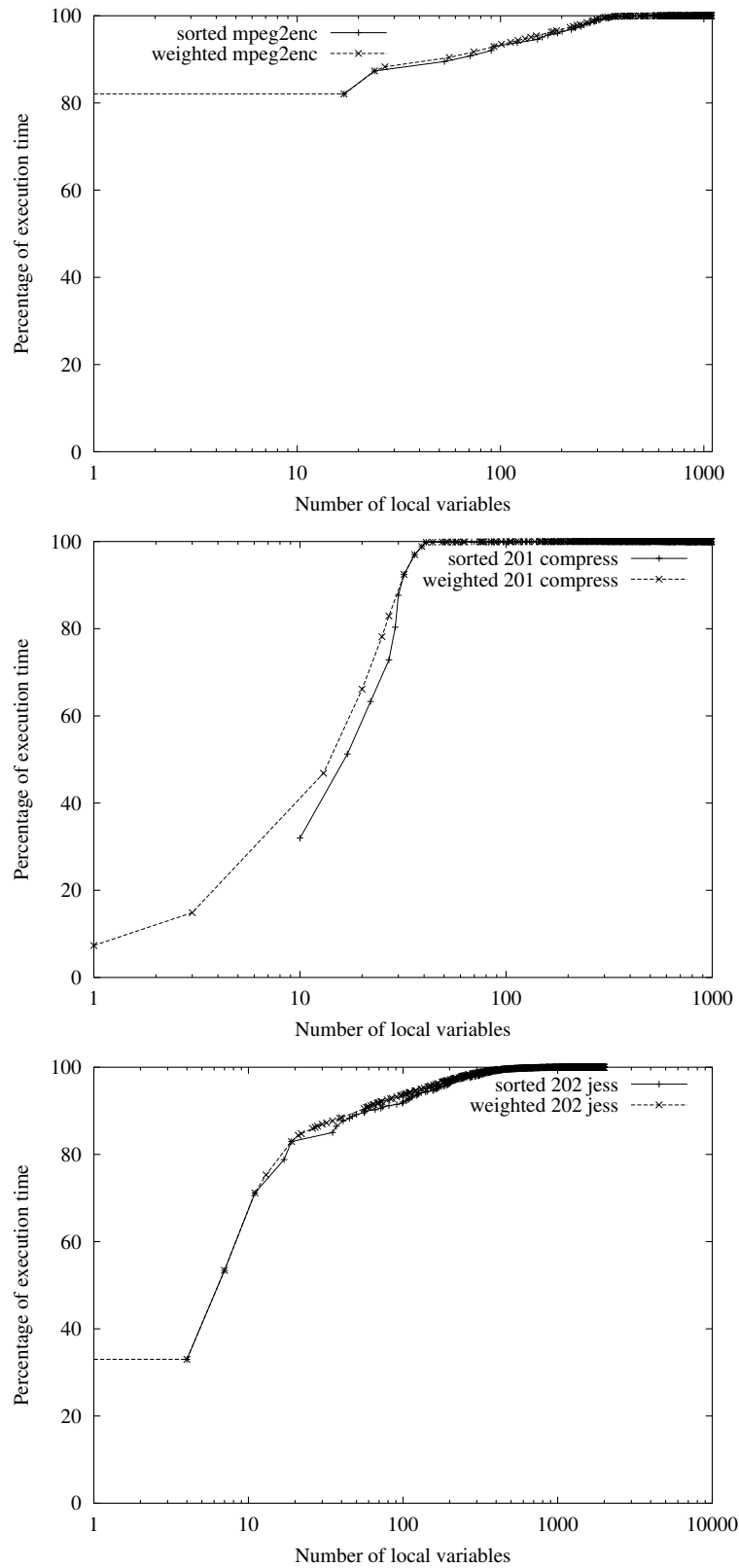


Figure 6.6: Approximate number of registers capturing a percentage of a total program execution

# Chapter 7

## Recursion

The last chapter described the fixed and sliding register window schemes that allow optimisation over method boundaries within the Dynamite JVM. The fixed register window scheme was chosen to be used by the Dynamite JVM. This chapter describes how recursion is dealt with in the fixed register window scheme, but the topic and solution are equally as appropriate for the sliding register window scheme.

Section 7.1 describes the approach to handling progressively more complex forms of recursion. Section 7.2 describes two approaches to handling stack fix-up within the Dynamite JVM once recursion has been detected. Section 7.2 then discusses the chosen scheme for the Dynamite JVM. Section 7.3 investigates the overhead of the chosen scheme. A comparison is performed of the Dynamite JVM to the HotSpot client and server JVMs on a recursive benchmark.

For the purposes of this thesis we provide the definition of recursion as definition 1.

**Def. 1** *Recursion is the calling of a method A from within a method B where method A either calls method B or another method called by method A calls method B. Method B may be the same as method A.*

### 7.1 Lazy recursion detection

Recursion is a problem for register window schemes in the Dynamite JVM as recursion requires a register window to be reused. If the Dynamite JVM were to allocate extra register windows in the abstract register pool for each recursive call

into a method, then there would need to be a corresponding translation of the method and the Dynamite JVM would have to pay a high translation penalty. There would also be code explosion only bounded by the maximum depth of the recursion.

This problem can be solved in a hardware register window schemes using a sliding or heap allocated register window that allows a certain amount of indirect addressing of registers. This hardware maybe available to a translation system, for example, it would be available if the SPARC processor were to be targeted by the Dynamite JVM's back-end. However, for instruction set architectures such as MIPS, IA32 and ARM it is unavailable and registers can only be directly addressed from within instructions. It is therefore assumed that the Dynamite JVM back-end's ISA registers can only be directly addressed.

For the fixed register window scheme to work, and not to pay a translation and code explosion penalty, it needs to deoptimise recursive code and revert it back to a stack managed in memory instead of registers. This deoptimisation need only apply to methods that would have their register window over-written by recursion. Non-recursive methods called by the recursive methods or methods that are not themselves recursive but call recursive methods need not save their stack frames.

Java has the ability to dynamically link classes as well as the ability to replace the primordial class loader replaced with one, for example, capable of loading class files from over the Internet. The ability to dynamically load and link code means a JVM can't perform an analysis on method calls, and the class hierarchy, that would be capable of detecting recursion statically (before the code has been executed).

Conventional method inlining approaches, which are analogous to the reasons for wanting a fixed register window scheme, use a memory stack initially and then optimise in cases where recursion does not exist. This scheme could be implemented within the Dynamite JVM, but it would require a new front-end optimisation phase added to the Dynamite JVM framework. The optimisation would work by using two different front-ends, one that creates dynamic code for the fixed register window scheme and one that produces for a memory stack. The front-end that produces code using the memory stack would be the one used in basic block mode. When a hot region is detected in the basic block control-of-flow graph and turned into a group block, the fixed register window

front-end would be called to retranslate the basic blocks belonging to the methods that are being optimised. The current Dynamite JVM is implemented to only have one optimising front-end, so considerable work would be necessary for this scheme. Chapter 9 considers how performance of the Dynamite JVM can be further improved.

As we are considering inlining not only direct method calls (such as calls to static, final or private methods), but also virtual method calls, it can never know whether the inlining optimisation has been performed safely. There is always the possibility that a new class will be loaded, or if the program executes in a different way, that the behaviour is altered and a previously non-recursive method is made recursive. An example of this is shown in figure 7.1.

```
// Define 2 classes where recursion does not exist
class OurObject {
    // ...
    public printName() {
        System.out.println (‘‘OurObject’’);
    }
    // ...
}

class UsesOurObject {
    // ...
    public OurObject x;
    public void printNames()
    {
        x.printName();
    }
    // ...
}

// Define a new class that introduces a potential for recursion
class ExtendsOurObject extends OurObject {
    // ...
    public UsesOurObject y;
    public void printName()
    {
        y.printNames();
    }
    // ...
}
```

Figure 7.1: Dynamic loading altering recursive behaviour of code

In figure 7.1 when classes `OurObject` and `UsesOurObject` are loaded no recursion exists. However, when `ExtendsOurObject` is loaded variable `x` in `UsesOurObject` can be of type `ExtendsOurObject`. This means the method `printNames` in class `UsesOurObject` can now call the method `printName` in `ExtendsOurObject` that calls `printNames` in class `UsesOurObject`, so recursion has potentially been introduced.

How the Dynamite JVM handles different kinds of recursion is described in sections 7.1.1, 7.1.2 and 7.1.3.

### 7.1.1 Single recursion

**Def. 2** *Single recursion: A method that recurses purely by statically calling itself.*

Single recursion is a programming technique using recursion deliberately to divide and conquer a problem. As the program is written to statically call the method that is currently being translated, detection of this kind of recursion is straightforward (the destination method equals the calling method).

In the Dynamite JVM when single recursion is detected, the translation of the recursive call needs to plant dynamic code to save the contents of the current register window to a stack. The stack pointer used to save the window is maintained in the first abstract register. The method call bytecode, currently being translated, has the suffix `_rec` added. The suffix stops recursion detection being performed if this bytecode is translated a second time. Also, the bytecode following the method call in the current method can easily tell where the window is, and if the `_rec` suffix is detected on the prior bytecode the window restored from memory.

The implementation is of a caller save scheme, whereby registers are saved to the stack when they are believed to be needed by a called method. This can create unnecessary stores to the memory stack when all the registers are not required by the called method. A callee save scheme would reduce the number of stores to the stack, however, a caller save scheme is simpler to implement. The general case overhead is minimal as the memory stack is a fallback for when the register window scheme can't be used.

To avoid unnecessary saving of local variables which are no longer used the translator scans the forthcoming bytecodes to see which local variables they use (simple liveness analysis within the method). If a backward branch is found it is

assumed all local variables are used. An example of this optimisation is shown in figure 7.2.

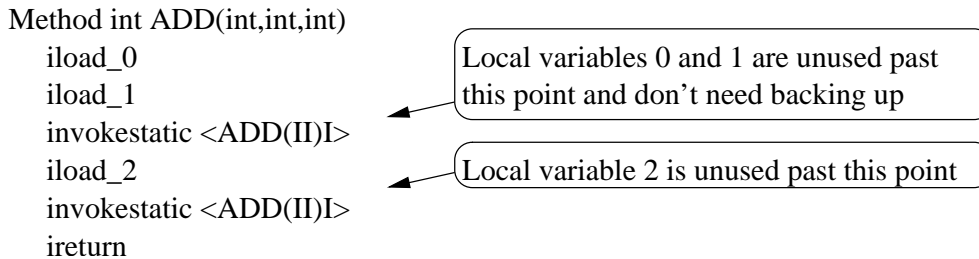


Figure 7.2: Reducing locals saved to stack

### 7.1.2 Direct mutual-recursion

**Def. 3** *mutual-recursion*: Method *A* which statically calls a method *B* that in turn statically calls method *A*.

Mutual-recursion is another programming technique that uses a number of recursive methods to divide and conquer a problem. As the method performing the call isn't the one being translated, detection of the recursion is more difficult. To solve this within the Dynamite JVM each method descriptor has a collection of methods called by the method. This set is added to each time a new method is called. By looking at the called methods the translator can determine if there is a path back to this method.

It is necessary to perform a stack fix-up so that the method being called has its register window appearing on the stack before the register window belonging to the method that is currently being translated. An example is shown in figure 7.3. The `put` and `rehash` methods are mutually recursive. Without fix-up the values of `this`, `key` and `value` in the `put` method, the first time the `rehash` method is called by `put`, will be overwritten by the second call to `put` from within the `rehash` method. The frame data will also be overwritten making the return address invalid. Section 7.2.1 and section 7.2.2 describe alternatives for the Dynamite JVM's implementation of stack fix-up.

As well as fixing up code the translator needs to ensure all translations of the previous method are invalidated so that a memory stack version can be translated in their place. As the Dynamite JVM only translates on demand the translation of the deoptimised code only occurs on code that will be executed after the recursion is detected.

```

/* Hashtable objects are used widely throughout the class library and by
user programs. This example doesn't provide the performance or features of
the class library implementation. */
class Hashtable
{
    /* The size of the hashtable - initialised to 0 */
    private int size;
    /* Number of entries in the hashtable - initialised to 0 */
    private int numEntries;
    /* The entries in the table (combined form a bucket) */
    private Object[] entry_values;
    private Object[] entry_keys;
    ...
    /* A Hashtable constructor */
    public void Hashtable (int initialSize)
    {
        size = initialSize;
        entry_values = new Object[size];
        entry_keys = new Object[size];
    }
    /* The rehash method increases the size of the hashtable */
    private void rehash()
    {
        int ctr;
        /* Create a new hashtable of greater size */
        Hashtable newHashtable = new Hashtable(size+100);
        /* Put the old hashtable elements into the new table */
        for (ctr=0; ctr < size; ctr++) {
            if (entry_keys[ctr] != NULL) {
                newHashtable.put(entry_keys[ctr], entry_values[ctr]);
            }
        }
        /* Make the new hashtable the one used */
        size = newHashtable.size;
        numEntries = newHashtable.numEntries;
        entry_values = newHashtable.entry_values;
        entry_keys = newHashtable.entry_keys;
    }
    /* Place a value at the location given by the key */
    final public void put(Object key, Object value)
    {
        /* Is the table full or is there a collision ? */
        while ((numEntries == size) || (entry_keys[key.hashCode()%size] != NULL))
        { /* Grow the table */
            rehash();
        }
        /* Insert the entry - all objects have a hashCode method */
        entry_values[key.hashCode() % size] = value;
        entry_keys[key.hashCode() % size] = key;
        /* Increase the number of entries */
        numEntries++;
    }
    ...
}

```

Figure 7.3: An example showing the need of recursion fix-up

### 7.1.3 Indirect mutual-recursion

**Def. 4** *Indirect mutual-recursion: A method A that indirectly calls another method B that calls method A.*

Complex recursion encompasses all other occasions a method ends up being called from within itself. An example is shown in figure 7.4. The `hashCode` method calls the `hashCode` method of the objects in the list. If a list of lists were created the `hashCode` method would be recursive.

```

/* This example class maintains a list */
class List
{
    ...
    /* Create the hashCode for this list - standard definition */
    public int hashCode()
    {
        /* Iterate down list combining hashCodes of each element */
        hashCode = 1;
        Iterator i = list.iterator();
        while (i.hasNext()) {
            Object obj = i.next();
            hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
        }
    }
    ...
}

```

Figure 7.4: Complex recursive example

The detection of complex recursion requires that the set of called methods in a method's descriptor to include not only static calls but virtual calls too. A virtual call calls a super classes method by default, but this may be overridden. The method called is determined by the virtual method table pointed to by an object's header. A virtual method table is created whenever a class is loaded. As a new class may be loaded it is not always possible for a complex recursive method to determine initially whether or not they are recursive.

The Dynamite JVM calculates the possible destinations of virtual call by examining the current class hierarchy at the point when the translation takes place. The Dynamite JVM also adds the called method to the descriptor of the previous method when it translates the fix-up code for the method. As all fix-up

code blocks are specialised to the calling method, all possible methods that are called by a method are known to the translator.

It is a feature of the basic block that performs the method call where the unknown recursion exists that it has not been translated. If it had been then the method would exist in a method's descriptor and the recursion would be detectable. As this method call is translated it is added to the descriptor of the calling method. If a call is translated within the method that is potentially recursive there will be a path from the method being translated back to itself. The translator at this point has detected the recursion and can invalidate previous translations and perform stack fix-up.

Other than the deoptimisation cost of retranslating code, the cost of this lazy recursion detection system is maintaining the list of called methods. As this is done only once per translation of a block containing a method call, the operation is considered cheap to perform.

## 7.2 Fixing the call stack

For the lazy recursion detection described in the last section to work in the general case, it requires a mechanism for the call stack to be fixed up. A fix-up involves placing a stack frame from a register frame into the memory stack frame, at the correct location, when recursion is detected. Two approaches are considered, one for immediate fixing of the memory stack and another for delayed fix-up.

### 7.2.1 Immediate stack fix-up technique

Immediate stack fix-up involves detecting which methods frames should be on the call stack. This is done by walking down the stack and seeing if any of the methods that have called the method we are in can be called by the method we are in. As the JVM must support a stack trace facility for exceptions, all the information for the stack walk are contained within the call stack. The JVM is able to distinguish memory stack from register window stack frames as the value of the memory stack's link pointers<sup>1</sup> are greater than the number of registers used in the register window scheme.

---

<sup>1</sup>Each call stack frame within the Dynamite JVM has a frame data region that holds a link value which points to the previous stack-frame data. The use of the frame data for exceptions is described in section 5.2.

After walking down the stack collecting information about what methods can be called by the current method, the stack is walked back up. Frames that can be called by the current method are inserted into the memory stack, the memory stack pointer and link pointer of the next frame are adjusted to reflect the change. Recursive methods detected whilst walking the stack have their translations invalidated so they are retranslated to use the memory stack.

The cost of performing the stack fix-up is proportional to the depth of the call stack. However, once the fix-up has been performed the code can continue to be executed without requiring a further retranslation in the future, unlike the delayed stack fix-up technique described in the next section.

### 7.2.2 Delayed stack fix-up technique

Delayed stack fix-up is a technique that dynamically alters target code to cause stack fix-up to occur just before the required frame is loaded from the memory stack. This prevents stack traversal that could potentially prove costly. As stack frames will be overwritten they are copied to backup frames that are stored into the memory stack when the stack is unwound. Figure 7.5 shows an example translation of a mutually recursive piece of code using this technique.

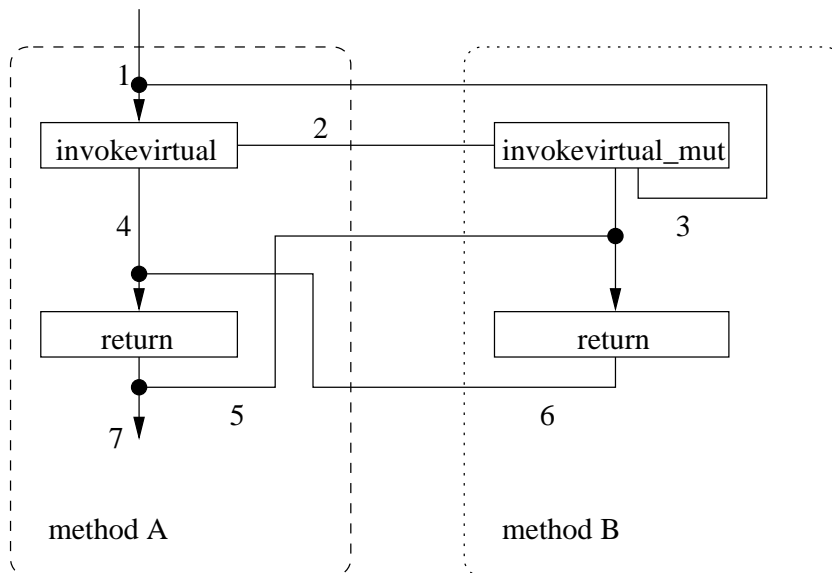


Figure 7.5: Delayed stack fix-up

The following is a description of the steps taken in figure 7.5:

1. Method A is entered and translated/executed until it reaches an `invokevirtual` bytecode.
2. Method B is entered and translated/executed until it reaches a recursive call to method A. The call is known to be recursive by checking the called methods list of method A and seeing that method B is reachable by it. The translation invalidates the translated basic blocks of method A as they now possess calls that need to be made recursive. Method A's stack frame is copied into a backup frame and the current bytecode altered to a `invokevirtual_mut` bytecode. The target code increments a counter which is used to calculate when the backed up frames should be restored.
3. Method A is entered again and translated/executed. If the call to method B were translated again it would be turned into a `invokevirtual_rec` bytecode and the memory stack used.
4. Method A is executed until it exits with a `return` bytecode.
5. As part of the fix-up caused by a return, method B discovers the previous bytecode is an `invokevirtual_mut` bytecode and inspects whether the counter has a value of 0. If it has then method A's frame is restored from the backup. The basic block containing the `invokevirtual_mut` is invalidated and the bytecode altered to a `invokevirtual_rec` as a counter no-longer needs maintaining.
6. Method B returns to method A.
7. Method A returns.

The test for a mutual-recursion counter hitting zero to invalidate blocks and reload the backed up frames requires a substitute call. Compatibility tests can't be used as if a compatibility test passes once it must always pass. The Dynamite JVM kernel can be altered to test compatibility based on counters, but it does not currently do so. To support threading in the kernel and to allow a re-entrant JVM, multiple backups of a method's frame are required for the different possible translations.

### 7.2.3 Chosen scheme

The runtime cost of the immediate stack fix-up is the cost to move stack frames up when one needs inserting lower in the stack. A delayed scheme requires multiple counters and backup frames to be kept as well as retranslation of basic blocks. The immediate stack fix-up scheme is currently implemented in the Dynamite JVM as it was expected that most recursion will be detected on shallow call stacks. A pathological case for the immediate fix-up scheme is shown in figure 7.6.

```
class test {  
public static int foo (bool val)  
{  
    if (val == true)  
        return bar(1000000);  
    else  
        return 1;  
}  
  
public static int bar (int val)  
{  
    if (val > 1)  
        return 1+bar(val-1);  
    else  
        return foo(false);  
}  
}
```

Figure 7.6: Pathological case for immediate stack fix-up

If the method `foo` is called with a value of `true` in figure 7.6 then the memory-stack will contain 1,000,000 copies of method `bar`'s frame when mutual-recursion is detected by the call back to method `foo`. The immediate stack fix-up scheme has to shuffle these 1,000,000 entries up the stack to insert method `foo`'s frame, whereas the delayed fix-up scheme would insert the frame when the stack had been unwound and therefore not incurring a copying overhead.

### 7.2.4 Example stack fix-up

An example of stack fix-up performed in the Dynamite JVM is shown in figure 7.7, it shows the stack before and after stack fix-up in one instance of a stack

fix-up performed in the `_202_jess` benchmark. The bytecode that calls out tells whether a frame is in registers or memory. Bytecodes with a `_rec` extension save the callers frame to memory before the call is performed. The descriptor for a method is also shown. The descriptor allows name polymorphism dependent on the parameters passed. The parameters are shown between brackets, the value to the right is the return type. Table 7.1 explains the meaning of the abbreviations.

I	Integer
J	Long
Z	Boolean
B	Byte
V	Void
S	Short
F	Float
D	Double
[x	Array of elements of type x
L..;	An object reference to a class specified between the L and the ;

Table 7.1: Descriptor abbreviation descriptions

In the example the method `spec/harness/Context.appendWindow` detects recursion and the methods from depth 14 down to depth 25 need fix-up performing on them. The methods at depths 16 and 17 already have there frames in memory so these frames are shuffled up and the frames that should be above and below them are inserted.

## 7.3 Performance

This sections looks at the performance of the lazy recursion and immediate stack fix-up technique. Section 7.3.1 examines how much the memory stack is used. Section 7.3.2 looks at how frequently stack fix-up is required and what the maximum cost of immediate fix-up will be. Section 7.3.3 analyses the hot loop of a recursive benchmark and compares the performance to the HotSpot JVM.

### 7.3.1 Cost of recursion

In section 6.4.1 a set of Java benchmarks were introduced which are used as examples to characterise the execution of Java programs. Table 7.2 shows the number of recursive instructions translated and executed in these benchmarks.

Depth	Method	Descriptor	Calls out with before	Calls out with after
25	spec/harness/Context.appendWindow	(Ljava/lang/String;) V		
24	spec/io/ConsoleOutputStream.flush	() V	invokestatic	invokestatic_rec
23	spec/io/ConsoleOutputStream.write	() V	invokevirtual	invokevirtual_rec
22	spec/io/ValidityCheckOutputStream.write	(I) V	invokespecial	invokespecial_rec
21	spec/io/ConsoleOutputStream.write	(B I I) V	invokevirtual	invokevirtual_rec
20	spec/io/ConsoleOutputStream.write	(B) V	invokevirtual	invokevirtual_rec
19	gnu/java/io/encode/EncoderEightBitLookup.write	([C I I) V	invokevirtual	invokevirtual_rec
18	java/io/Writer.write	(Ljava/lang/String; I I) V	invokevirtual_rec	invokevirtual_rec
17	java/io/OutputStreamWriter.write	(Ljava/lang/String; I I) V	invokevirtual_rec	invokevirtual_rec
16	java/io/PrintWriter.write	(Ljava/lang/String; I I) V	invokevirtual	invokevirtual_rec
15	java/io/PrintWriter.write	(Ljava/lang/String;) V	invokevirtual	invokevirtual
14	java/io/PrintWriter.print	(Ljava/lang/String;) V	invokevirtual	invokevirtual
13	java/io/PrintWriter.println	(Ljava/lang/String;) V	invokevirtual	invokevirtual
12	java/io/PrintStream.println	(Ljava/lang/String;) Z	invokevirtual	invokevirtual
11	spec/benchmarks/_202_jess/Jess.run_jess	(Ljava/lang/String;) J	invokevirtual	invokevirtual
10	spec/benchmarks/_202_jess/Jess.inst.main	(Ljava/lang/String;) J	invokevirtual	invokevirtual
9	spec/benchmarks/_202_jess/Main.runBenchmark	(Ljava/lang/String;) J	invokevirtual	invokevirtual
8	spec/benchmarks/_202_jess/Main.harnessMain	(Ljava/lang/Object;) J	invokestatic	invokestatic
7	spec/harness/ProgramRunner.runOnce	(Ljava/lang/Object; I J I Ljava/util/Properties;) Lspec/harness/BenchmarkTime;	invokeinterface	invokeinterface
6	spec/harness/ProgramRunner.runBenchmark2	() Ljava/util/Properties;	invokespecial	invokespecial
5	spec/harness/ProgramRunner.runBenchmark	() V	invokevirtual	invokevirtual
4	spec/harness/ProgramRunner.run	() V	invokevirtual	invokevirtual
3	spec/harness/RunProgram.run	(Ljava/lang/String; Z Ljava/util/Properties;) Lspec/harness/BenchmarkDone; V		
2	SpecApplication.runBenchmark	(Ljava/lang/String;) V	invokestatic	invokestatic
1	SpecApplication.main	(Ljava/lang/String;) V	invokestatic	invokestatic

Figure 7.7: Example of stack fix-up

benchmark	invoke- virtual trans- lated	invoke- virtual exe- cuted	invoke- special trans- lated	invoke- special exe- cuted	invoke- static trans- lated	invoke- static exe- cuted
helloworld	4	4	0	0	0	0
mpeg2dec	4	16	0	0	0	0
mpeg2enc	4	40	0	0	0	0
_201_compress	18	778	3	299	1	7
_202_jess	54	76742	10	11887	9	12144

Table 7.2: Recursive instruction counts

benchmark	invoke- virtual trans- lated	invoke- virtual exe- cuted	invoke- special trans- lated	invoke- special exe- cuted	invoke- static trans- lated	invoke- static exe- cuted
helloworld	9.756%	4.255%	0%	0%	0%	0%
mpeg2dec	2.381%	0.177%	0%	0%	0%	0%
mpeg2enc	2.174%	0.001%	0%	0%	0%	0%
_201_compress	3.462%	0.005%	0.325%	0.012%	0.397%	0.055%
_202_jess	3.617%	1.339%	0.791%	4.523%	1.815%	9.576%

Table 7.3: Recursive instructions as a percentage of their base instruction

Although the \_202\_jess benchmark executes fewer bytecodes than the mpeg2dec, mpeg2enc and \_201\_compress benchmarks it performs more recursion. Table 7.3 shows what percentage of the respective invoke bytecodes these recursive bytecodes make up.

Table 7.4 shows the percentage of recursive invoke bytecodes as part of the total translated and executed bytecode mix.

Table 7.5 shows the total number of values placed on the memory stack by

benchmark	invoke- virtual trans- lated	invoke- virtual exe- cuted	invoke- special trans- lated	invoke- special exe- cuted	invoke- static trans- lated	invoke- static exe- cuted
helloworld	0.091%	0.018%	0%	0%	0%	0%
mpeg2dec	0.015%	0.000%	0%	0%	0%	0%
mpeg2enc	0.012%	0.000%	0%	0%	0%	0%
_201_compress	0.092%	0.000%	0.015%	0.000%	0.005%	0.000%
_202_jess	0.159%	0.052%	0.029%	0.008%	0.027%	0.008%

Table 7.4: Recursive bytecodes as a percentage of the total instruction mix

benchmark	invokevirtual total	invokespecial total	invokestatic total
helloworld	20	0	0
mpeg2dec	80	0	0
mpeg2enc	200	0	0
_201_compress	4511	1065	28
_202_jess	670083	70206	90008

Table 7.5: The total size of frames saved and loaded from the memory stack by recursive invoke bytecodes

benchmark	invokevirtual av- erage frame size	invokespecial av- erage frame size	invokestatic av- erage frame size
helloworld	5	0	0
mpeg2dec	5	0	0
mpeg2enc	5	0	0
_201_compress	5.798	3.562	4
_202_jess	8.732	5.906	7.412

Table 7.6: The average size of frames saved to the memory stack by recursive invoke bytecodes

recursive bytecodes. Table 7.6 uses these figures to calculate the average frame size stored by a recursive bytecode. Each stack value is a 32bit word.

Table 7.6 shows that the average memory stack frame size is between 3 and 9 words. The frame data required for a memory stack entry is 3 words, these hold the method associated with this frame, a link value to the next frame and the return address of the method. Other frame locations hold local variable values that will be used after the method has returned and any values left on the calling method's stack before the call was performed.

The results of this section have shown that recursive invoke bytecodes make up less than 10% of the respective invoke bytecodes, that are translated and executed by the Dynamite JVM, on the chosen benchmarks. The recursive bytecodes make up less than 0.16% of the overall total instruction mix. The number of parameters copied to and from the memory stack is relatively small, fewer than 9 words. The execution time performance of the technique is measured in section 7.3.3. The following section examines the performance of the immediate stack fix-up scheme.

### 7.3.2 Cost of stack fix-up

The Dynamite JVM was instrumented to record the number of stack fix-ups performed. As the cost of stack fix-up is related to the depth of the stack, the stack depth for each fix-up was added to a cumulative total. A cumulative total of all the maximum memory stack frame sizes was also recorded. The results are shown in table 7.7.

benchmark	stack fix-up operations	cumulative stack depth	cumulative frame sizes
helloworld	4	24	90
mpeg2dec	4	52	326
mpeg2enc	4	28	166
_201_compress	13	138	846
_202_jess	35	710	4538

Table 7.7: Stack fix-up cost

Table 7.7 shows fewer recursive stack fix-up operations being performed than translated recursive bytecodes shown in table 7.2. Stack fix-up is only necessary when recursion is detected and not when recursive invoke bytecodes are re-translated. Re-translation is necessary when the Dynamite JVM believes a translation has become unsafe through a potential recursion being introduced. As we know these particular bytecodes, and from that the containing basic block, are recursion safe (they are using the memory stack instead of the fixed register window) a potential optimisation may be to avoid invalidating these blocks.

The cumulative stack depth and frame sizes show that for the \_202\_jess benchmark 710 stack entries are examined and in the worst case 4538 stack frame entries (each a 32bit word) would need moving to perform a fix-up operation of inserting a frame in to the base of the memory stack (for each of the 35 performed fix-up operations). In practice most fix-up occurs at the top of the stack, as this is where the current working set of methods are. Even so the worst case performance of immediate stack fix-up won't be large as copying regions of memory is relatively cheap compared to the re-translation costs associated with the delayed stack fix-up scheme.

### 7.3.3 Performance comparison

The Takeuchi function [Knu91] is a heavily recursive function widely used to test recursive performance in Lisp environments. A Java version is shown in figure 7.8.

```
class takeuchi
{
    public static int result;
    public static int tak(int x, int y, int z) {
        if (y >= x)
            return z;
        else
            return tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y));
    }
    public static void main(String args[]) {
        result = tak(6,0,69);
    }
}
```

Figure 7.8: Takeuchi benchmark program

The method `tak` that performs the Takeuchi function takes three arguments and a result is computed recursively. The method is declared statically as it does not act on any instance data. The `main` method is the 1st method called by the JVM, in the benchmark it calls the function `tak` with three arguments which are decided at compile time.

Figure 7.9 shows the bytecodes of the `tak` method as a basic block graph. Figure 7.10 shows the Dynamite JVM basic block graph and IR for the `tak` method.

The bytecodes turn into more IR than is typical of a Java program as there are frequent method calls in the `tak` method, and this is a relatively complex bytecode. The labels in the top left of the IR are the addresses of the Java program's basic blocks in memory. Following this is a list of stores that need to be performed by this basic block, if any. Following this is a list of register definitions within this basic block. Successors are indicated on the lines leaving the blocks.

The store lists of basic blocks 0x08150867 and 0x0815086f have the stores to the memory stack backing up the current method's local variables. Basic blocks 0x0815087f and 0x08150882 avoid storing local variables to the memory stack as local variables are not used in the basic blocks following these. Following the

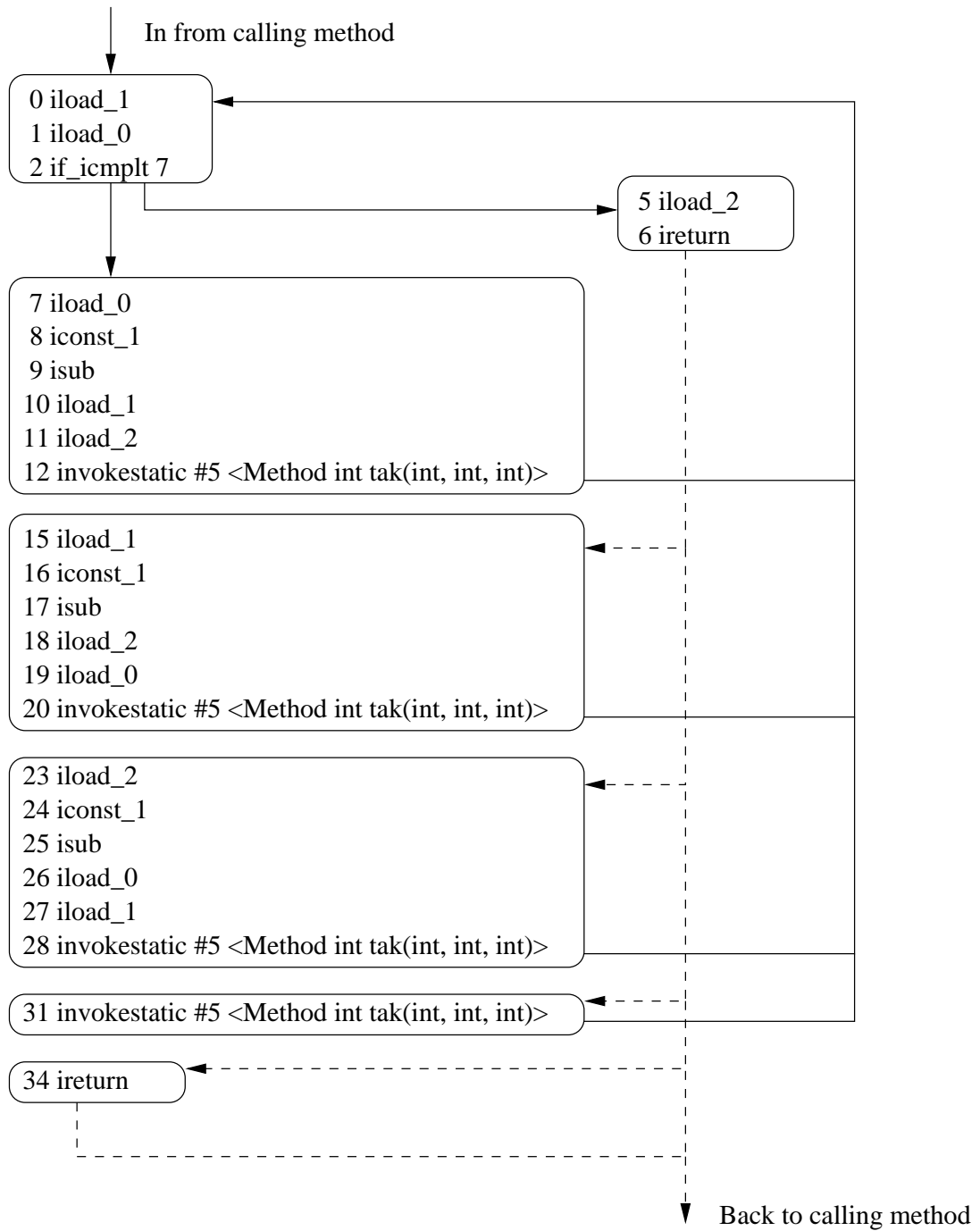


Figure 7.9: Takeuchi method bytecode basic block graph

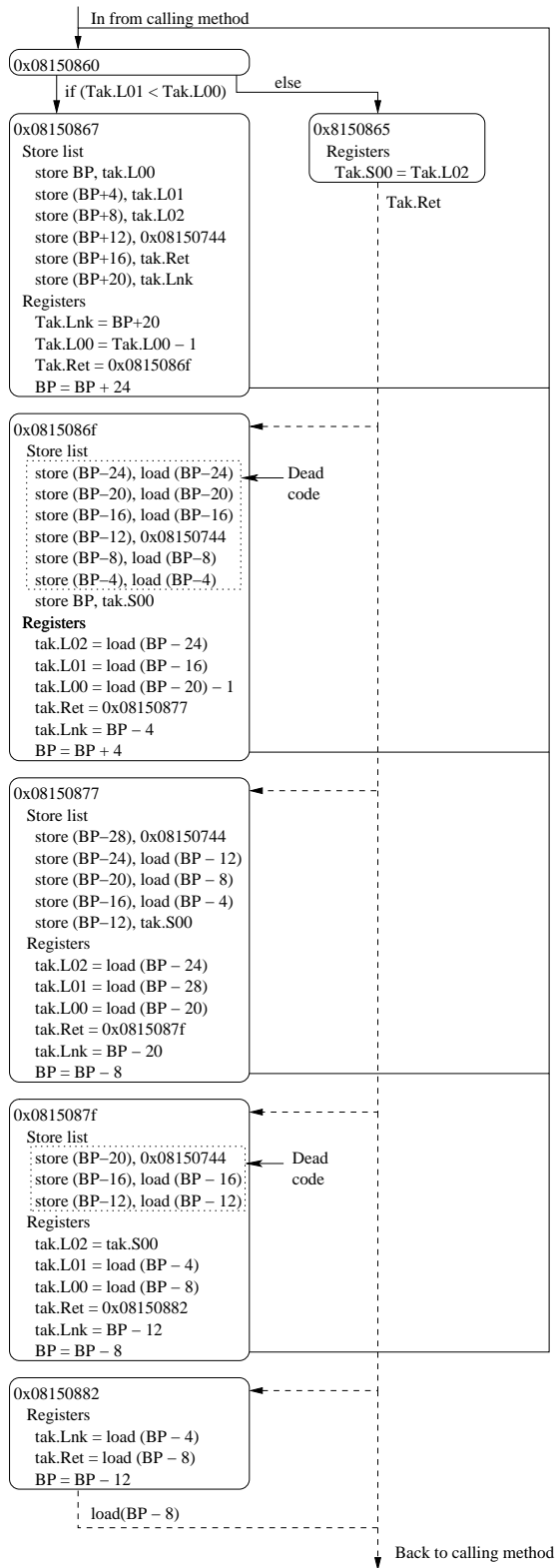


Figure 7.10: Takeuchi function translated into IR

local variables in the store list comes the frame data and then any stack values that need preserving.

In basic block 0x0815086f the stack frame is loaded at the beginning of the basic block and stored again at the end with the local variables and frame data being unchanged and at the same position in the memory stack. This is dead code (highlighted in figure 7.10) that can be eliminated by the Dynamite JVM. Likewise, the frame data storing highlighted in basic block 0x0815087f can be eliminated.

Figure 7.11 and figure 7.12 show the performance of the Dynamite JVM, the Dynamite JVM with group block optimisations (Dynamite -O), the HotSpot client JVM version 1.4 and the HotSpot server JVM version 1.4 [Sun99b]. Figure 7.11 shows the performance of the JVMs at executing a small number of bytecodes, figure 7.12 shows the performance of the JVMs at executing a larger number of bytecodes. All tests were run on an unloaded AMD Athlon 1GHz machine with 256MB RAM running SuSE Linux 7.2 and times recorded using the time [KMJP00] command<sup>2</sup>.

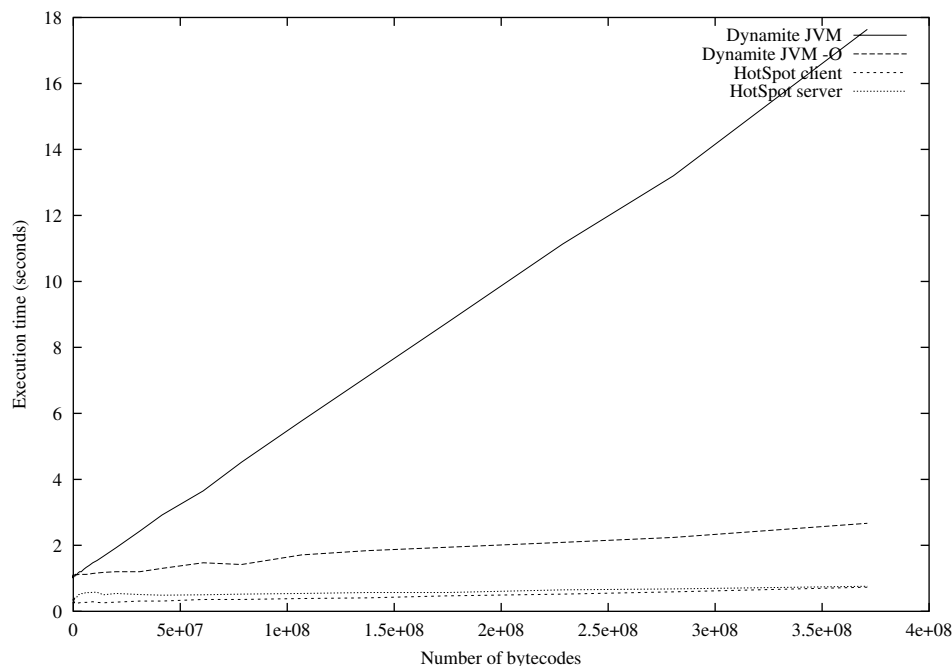


Figure 7.11: Takeuchi execution results

<sup>2</sup>The tests were run in single user mode and the minimum user time of 10 runs was recorded to try and cancel out operating system and architectural overheads caused by other processes being run on the test machine.

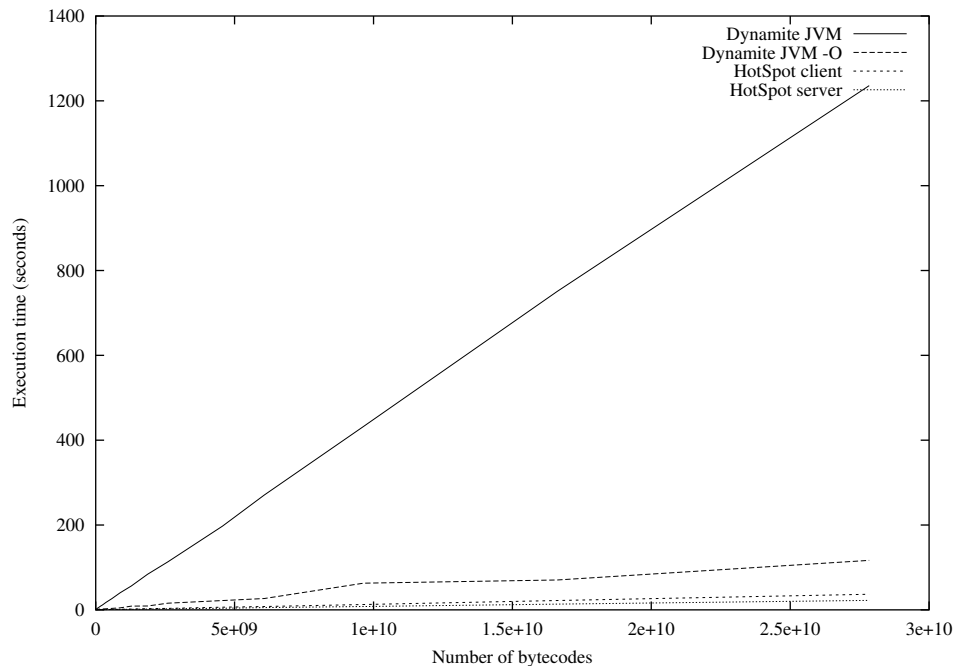


Figure 7.12: Takeuchi execution results (more bytecodes)

benchmark	bytecodes per second
Dynamite JVM	23,146,257
Dynamite JVM -O	243,030,690
HotSpot client JVM	768,709,402
HotSpot server JVM	1,238,759,081

The number of bytecodes executed by a particular run of the JVM was determined using the table included in appendix section B.3. The complexity of the Takeuchi function means the amount of recursion and bytecodes executed is not proportional to its input parameters.

The results show that the Dynamite JVM has a larger start-up cost than the HotSpot client and server JVM. Without group block optimisations the overhead of the Dynamite JVM's control loop is apparent. With group block optimisation, performance is improved as shown by the improved execution time with respect to the number of bytecodes. Table 7.3.3 shows the number of bytecodes a second executed by each JVM with the start-up overhead removed.

Performance is down with the Dynamite JVM due to load-store dead code elimination not being performed. Performance is considered further in the next chapter.

A multi-phase optimisation scheme is used in the HotSpot [Sun99b] JVM. De-optimisation is performed when optimisations have proved unsafe. An example of this would be when method inlining has been performed on a method that cannot be inlined due to a new class being loaded. The techniques used in the Dynamite JVM differ in that recursion is assumed not to exist when the 1st translation takes place. As HotSpot uses an interpreter it will know of most mutual and complex recursion when it comes to translate, whereas the Dynamite JVM only detects single recursion.

## 7.4 Summary

This chapter has presented the problem and solutions to recursion detection and fix-up for the Dynamite JVM. Lazy recursion detection is used as a solution to detecting recursion as dynamic loading of classes means full program analysis is not possible at runtime. A consequence of detecting recursion late is the need to perform stack fix-up. An immediate and delayed fix-up technique are described with the immediate stack fix-up scheme used in the Dynamite JVM. Finally the chapter analysed the performance of the recursion schemes showing low overhead on typical benchmark programs, with recursive bytecodes making up less than 10% of the method call bytecode mix and the associated overhead being between 3 and 9 words of data to push and pop from the memory stack.

On the highly recursive Takeuchi benchmark performance was 3.163 times slower per executed bytecode than the HotSpot client JVM as analysed in section 7.3.3. There is clear evidence of potential optimisations in the Takeuchi benchmark, with two regions of dead code highlighted in the hot region of the IR graph (figure 7.10). However, performance is lower largely down because of the Dynamite JVM's kernel and back-end. This is examined further in the next chapter.

Lazy recursion detection, immediate stack fix-up and delayed stack fix-up are applicable to a sliding register window scheme. Sliding register windows and the fixed register window scheme, used throughout this chapter, are described in chapter 6.

## Chapter 8

# Overall System Performance

The thesis so far has presented the Dynamite JVM and discussed the optimisations performed within it. Chapter 6 introduced fixed register window allocation. It was demonstrated that inter-procedural optimisation with the fixed register window scheme requires little additional code translation and generation. In a basic block translator fix-up code accounted for less than 1.3% of the instruction mix and typically required fewer than 2 abstract register copies. With group block optimisations this copying overhead can be eliminated through dead code elimination.

A different scheme was suggested for inter-procedural optimisation, a sliding register scheme. With a conservative model of this scheme, it was calculated that between 1.819 and 5.103 translations were required per method.

It was also shown that although the fixed register window scheme works regardless of target architecture, it benefits from architectures with more registers. With around 100 registers a group block can be created, for the benchmark cases, that contains 80% of the total translated code and has no register spills or fills performed.

Chapter 7 showed that recursive method call bytecodes, necessary for the inter-procedural optimisation, make up less than 10% of all translated method calls. The associated cost of these bytecodes was stacking of between 3 to 9 words to and from memory. Recursive bytecodes are analogous to method call bytecodes in JVMs where the method calls haven't been inlined. This sub-optimal case was found to be only necessary in less than 10% of all executed method calls, and accounted for less than 0.16% of the overall instruction mix.

Chapter 7 measured the performance of the Dynamite JVM compared to

the HotSpot client and server JVMs [PVC01] on the highly recursive Takeuchi function. Performance was 3.163 times slower than the HotSpot client JVM and 5.097 times slower compared to the HotSpot server JVM. It was shown at a high-level that this benchmark could dead code eliminate many memory accesses.

This chapter continues the analysis of the performance of the Dynamite JVM. The following section describes the aspects of the Dynamite JVM that will be measured.

## 8.1 Break down of JVM performance

An equation for the total execution time of a Java program is shown in equation 8.1.

$$\text{Execution time} = \text{Start up time} + \text{Translation time} + \text{Bytecode execution time} \quad (8.1)$$

Every time a JVM is started data structures necessary for garbage collection and translation are created. Runtime objects such as threads and the string array passed into the main method are created. Not only must the objects be created but the classes for these objects must be initialised. This time varies depending on what classes are used in the bootstrapping process, but this time is independent of work done executing the main bytecodes of the program and will be referred to here as the start up time.

The translation time is the time spent translating bytecodes. In a translation system the translated bytecodes are cached so they are only translated, dependant on the JVM, the first time a class is loaded, a method called, an exception executed or when a basic block is executed. Hardware JVMs or JVMs that use an interpreter spend no time translating. JVMs such as the HotSpot server JVM [PVC01] heavily optimise certain bytecodes, spending time translating bytecodes more than once and thereby increasing their translation time. Section 8.2 shows the saving in the number of translated bytecodes achievable by the Dynamite JVM. Section 8.4 examines the time it takes the Dynamite JVM and HotSpot JVMs to translate a fixed number of bytecodes. The immaturity of the Dynamite JVM results in slow performance.

The bytecode execution time of the JVM varies approximately with how many bytecodes need to be executed. The number of bytecodes executed depends on

the program and the dataset it is working on. To a certain extent the libraries used by the JVM have an impact on the performance, commercial JVMs having carefully optimised libraries. Section 8.3 examines the time taken to execute a varying number of bytecodes. Again, a comparison of the Dynamite JVM and HotSpot JVMs is performed. The Dynamite JVM produces code approximately 5 times slower than the HotSpot JVMs, section 8.5 examines the code produced by the Dynamite JVM to demonstrate areas where performance can be increased.

Latency is an important aspect to a users experience of a JVM. Latency is the time between requesting an activity and that activity being performed. For example, if a menu button is clicked within an interactive application then the user would expect the menu to appear within an acceptable period of time. In the JVM latency is comprised of both translation and execution times. An interpreter or Java processor can execute bytecodes without any translation overhead, so the time to execute the initial bytecodes is low. A JIT cannot execute bytecodes until they have been translated, however, the execution time of the bytecodes is typically lower so that the time spent translating is made up when the bytecodes are executed. Section 8.3 shows some of the latency characteristics of the Dynamite JVM and HotSpot JVMs.

The benchmark used to examine the properties of the Dynamite JVM and HotSpot client and server JVMs is a Java port of the Dhrystone benchmark [Oka02]. The Dhrystone benchmark was originally created for Ada [Wei84] and provides a Dhrystone MIPS comparison to a VAX 11/780. It is a synthetic benchmark designed to simulate realistic integer, string, memory and logic operations. As such it is a less than ideal example of an object-oriented Java program. It does, however, have the advantage of being small and easy to understand, making it ideal for experimentation. It also avoids known performance bottlenecks in the Dynamite JVM, such as floating point operations.

Ian Walsh, technical director for MIPS in Europe, said: “It’s very easy to say Dhrystone is not the best benchmark in the world but everyone knows the same tricks and does the same things and that makes it fair again.” [Fla01]

## 8.2 Translation saving

By translating basic blocks instead of methods (as in a JIT compiler) the number of translated bytecodes is reduced. The Dynamite JVM was instrumented to record the number of translated bytes in the translation loop and the number of bytes that would have been translated by a JIT compiler. Bytes were recorded rather than bytecodes for simplicity. The translated bytes were divided in to bytes that were translated in library and application code. Library code was distinguished by method identifiers beginning with either `java` or `gnu` (`gnu` objects are part of the Classpath library). The results are shown in table 8.1

Benchmark	JIT library	JIT non-library	Dynamite JVM library	Dynamite JVM non-library
helloworld	7,407	9	6,815	9
dhystone	11,174	1,155	9,655	1,114
mpeg2dec	9,501	31,715	8,523	27,329
mpeg2enc	9,494	39,693	8,509	35,587
_201_compress	14,510	15,026	12,090	12,666
_202_jess	17,595	34,446	14,328	28,195

Table 8.1: Number of translated bytes of a JIT compiler and the Dynamite JVM

Table 8.2 shows the percentage reduction achieved by basic block translation.

Benchmark	library saving	non-library saving
helloworld	7.99%	0%
dhystone	13.59%	3.55%
mpeg2dec	10.29%	13.83%
mpeg2enc	10.37%	10.34%
_201_compress	16.68%	12.70%
_202_jess	18.57%	18.14%

Table 8.2: Reduction in translated bytes by basic block compilation

Table 8.2 shows the Dynamite JVM translates fewer bytes of code than a JIT compiler. A JIT compiler does not necessarily compile all the methods that are loaded, only those that are executed. Comparing the amount of translation performed by the Dynamite JVM to a JIT compiler is therefore dissimilar to comparing the amount of translation performed by a dynamic binary translator to a static translator, as a static translator must compile the whole of a binary file.

### 8.3 Performance of translated code

The execution time of the Dhrystone benchmark is dependent on the number of times the main loop is executed. This is shown in equation 8.2, where  $S$  is the start-up time,  $T$  is the translation time,  $B$  is the bytecode execution time excluding the loop execution time,  $E$  is the number of times the loop is executed and  $BL$  is the execution time for each loop.

$$\text{Execution time} = S + T + B + E * BL \quad (8.2)$$

For a JVM with two translation stages, such as the Dynamite JVM, the loop is translated, executed a certain number of times, translated for a second time and then this translation executed until the loop count is reached. The HotSpot JVMs have three translation stages with each translation phase being triggered by the loop being executed a certain number of times. This is shown in equation 8.3, where  $E1$  to  $E3$  are the number of times the results of each translation pass are executed and  $BL1$  to  $BL3$  are the respective execution times. The value of  $T$  is the translation time of all three translation passes. The value of  $E$  in equation 8.2 is the sum of  $E1$ ,  $E2$  and  $E3$ .

$$\text{Execution time} = S + T + B + E1 * BL1 + E2 * BL2 + E3 * BL3 \quad (8.3)$$

For large values of  $E$ ,  $E1$  and  $E2$  will be comparatively small and constant as  $E$  is increased. By increasing  $E$  the execution time will increase and the rate of change of which will be  $BL$ . For a JVM with three translation phases the rate of change will be  $BL3$ . The Dynamite JVM fits equation 8.3 with a value of 0 for  $E1$ .

#### 8.3.1 Externally measured performance

A harness was written that ran the Dhrystone benchmark for a set number of loops measuring the time taken to execute. All tests were run on an unloaded AMD Athlon 1GHz machine with 256MB RAM running SuSE Linux 7.2 and times recorded using the time [KMJP00] command to record user mode execution time. Section 8.3.2 measures the same performance using a measurement generated by the Dhrystone benchmark.

Figures 8.1, 8.2 and 8.3 show the execution times of the Dhrystone benchmark for varying value of  $E$ . Each test was repeated 25 times and the lowest value used. The value of  $E$  was increased in increments of 1 up until 100, then increments of 100 up until 70,000, then in increments of 1,000 up until 300,000 and finally in increments of 100,000 up until 2,800,000.

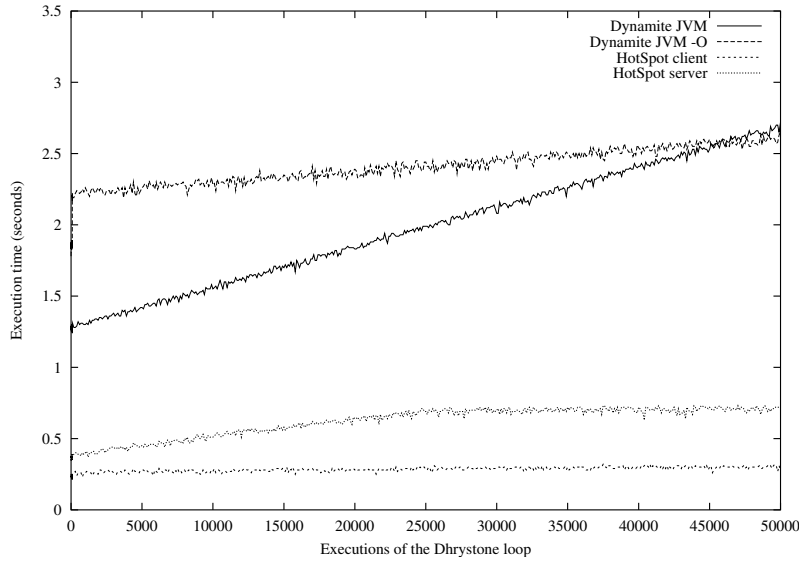
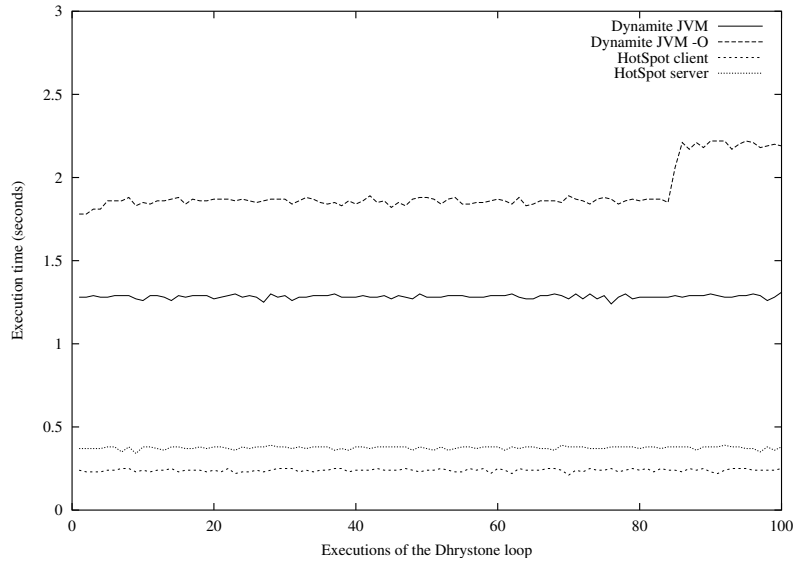
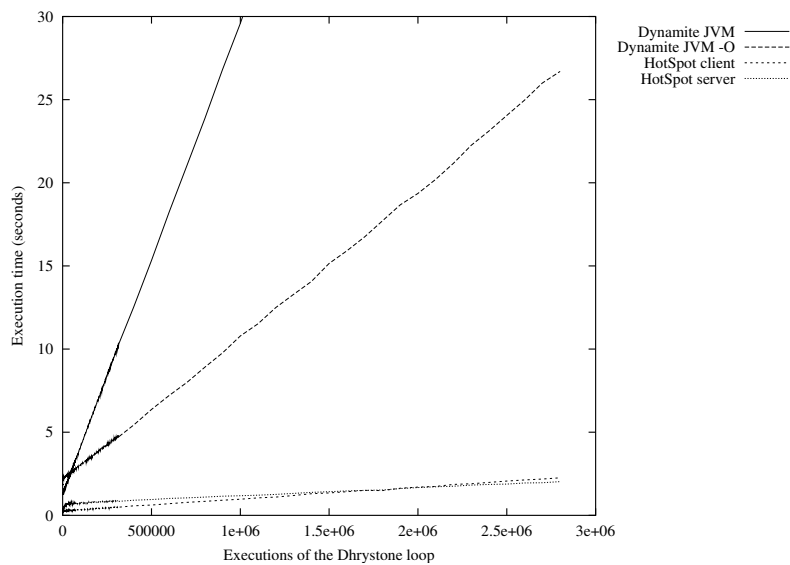


Figure 8.1: Execution cost: total execution time for values of  $E$  up to 50,000

Figure 8.2 shows the extra translation cost when group block optimisation is performed after 85 iterations of the Dhrystone loop. Initial performance of the Dynamite JVM with group block optimisation is worse than without, as group block optimisation has been performed on native library code. Figure 8.1 shows that the extra translation cost of creating a group block within the Dynamite JVM has been cancelled out by the increased performance of the translated code by 45,000 executions of the Dhrystone loop. The HotSpot server JVM's execution time increases at a lower rate after the number of executions of the Dhrystone loop has passed 25,000. Figure 8.3 shows that the HotSpot server JVM's execution time is the same as or lower than the HotSpot client JVM after 1,700,000 executions of the Dhrystone loop.

### 8.3.2 Internally measured performance

The Dhrystone loop, executing on a particular JVM, will execute a certain number of bytecodes each time around the loop. By instrumenting the Dhrystone

Figure 8.2: Execution cost: total execution time for values of  $E$  up to 100Figure 8.3: Execution cost: total execution time for values of  $E$  up to 2,800,000

loop to emit the period of time between each pass an execution profile can be plotted. The starting time is a static initialised in the class initialiser. The `java.lang.System.currentTimeMillis` library call is used to get the time. The test machine is the same as in section 8.3.1, each test was repeated 20 times and the results with the lowest execution time were used. The results are shown in figure 8.4.

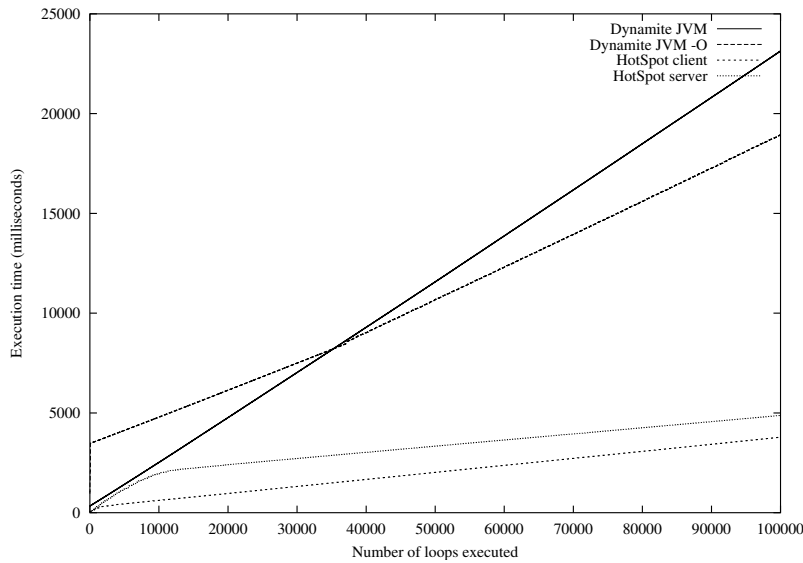


Figure 8.4: Internally measured execution cost

These results mirror those of section 8.3.1. The graph is smoother due to the method of generating the time values. The time values are larger because the timer is measuring the total execution time and not just time spent in user mode. As certain sections of the code aren't timed, translation time expended during termination of the program isn't incorporated into the graph. This benefits the group block optimising version of the Dynamite JVM, which has better performance than the Dynamite JVM without group blocks after 36,000 loop executions.

### 8.3.3 Analysis

The performance of the externally and internally measured performance varies greatly. Table 8.3 summarises the features of the graphs from the previous two sections.

JVM	External		Internal	
	Loops to reach optimal performance	Increase in time per Dhrystone loop	Loops to reach optimal performance	Increase in time per Dhrystone loop
Dynamite JVM	1	0.0298ms	1	0.228ms
Dynamite JVM -O	85	0.00889ms	88	0.155ms
HotSpot client	1	0.000710ms	7	0.0373ms
HotSpot server	25,000	0.000465ms	12,000	0.0311ms

Table 8.3: Summary of execution performance

Table 8.3 shows the difference the measurement mechanism makes. Performance is an order of magnitude better when user time is recorded instead of wall clock time, as recorded by the internal measurement. The optimal time to execute a Dhrystone loop with the Dynamite JVM with group block optimisations (-O) is 12.5 times slower than the HotSpot client JVM, with external timing. Similarly against the HotSpot server JVM the Dynamite JVM with group block optimisations is 19.1 times slower. However, with internal timing, the Dynamite JVM is 4.16 and 4.98 times slower than the HotSpot client and server JVMs respectively.

This performance is disappointing, however, the Dynamite JVM provides much simpler compilation than the HotSpot JVMs. Section 8.5 examines code produced by the Dynamite JVM and shows how it can be improved. The performance of the Dynamite JVM also suffers as the Dhrystone loop makes use of the `java.lang.String` library for string access and comparison. Optimisation of this library would increase the performance greatly and could be achieved by writing native methods for the String methods. More optimally the Dynamite JVM can recognise certain method calls and plant IR that performs the methods algorithm.

The results from this section have shown that simple compilation or interpretation are an advantage for improved latency up until large loop counts. The Dhrystone loop threshold for the Dynamite JVM with and without group block

optimisations is a loop count of either 45,000 when user time is measured externally or 36,000 when the wall clock time is measured internally. The external loop threshold is greater as a larger amount of more optimal code needs to be executed to recoup the performance lost by performing more complex translation on code outside of the inner loop, which in itself doesn't justify increased compilation effort.

## 8.4 Translation performance

Translation time increases as the number of bytecodes translated in a program increases. The Dhrystone benchmark is a large loop that is executed a number of times entered as a parameter to the program. The loop will only be translated once per run of the benchmark. However, the loop can be unrolled<sup>1</sup> inside a test harness, and by doing so the number of translated bytecodes increased. Equation 8.4 shows the execution time for a JIT compiler executing the Dhrystone benchmark, where  $S$  is the start-up time,  $B$  is the bytecode execution time,  $T$  is the translation time excluding the unrolled loop,  $N$  is the number of times the Dhrystone loop has been unrolled,  $TL$  is the time to translate a loop of the Dhrystone benchmark, and  $LO$  is a loop overhead or saving incurred by unrolling the loop.

$$\text{Execution time} = S + B + T + N * (TL + LO) \quad (8.4)$$

By keeping the number of executions of the Dhrystone loop constant the variables  $S$  and  $B$  will remain constant also. The value of  $LO$  is small in comparison to  $TL$ . So by varying  $N$ , the number of times the Dhrystone loop has been unrolled, and plotting this against the execution time, it is possible to calculate the value of  $TL$ , the time to translate a loop, by the gradient of the slope.

JIT performance is different to HotSpot and Dynamite JVM as it uses only a single translation pass. Equation 8.5 shows the execution time for a 2 pass translator, such as the Dynamite JVM, where the 1 or 2 suffix represents a translation, execution or loop overhead time associated with each translation pass.

---

<sup>1</sup>Loop unrolling is a conventional compiler optimisation where the loop body is repeated a number of times. This benefits the target architecture of the compiler with potentially reduced cache misses and branch mispredictions.

$$\text{Execution time} = S + B1 + B2 + T + N * (TL1 + TL2 + LO1 + LO2) \quad (8.5)$$

For large loop values the execution of second pass translated bytecodes ( $B2$ ) will be much larger than ( $B1$ ). For example, the Dynamite JVM performs second pass translation after 85 executions of the first pass code. If the loop is executed 10,000 times then the executed bytecodes from the first translation pass will account for less than 1% of the total executed bytecodes.

HotSpot uses a 3 pass translation and execution scheme [PVC01] where the 1st pass is actually an interpreter and therefore incurs no translation penalty. Equation 8.6 shows the total execution time of the unrolled Dhrystone running on HotSpot.

$$\text{Execution time} = S+B1+B2+B3+T+N*(TL2+TL3+L01+LO2+LO3) \quad (8.6)$$

So by varying the  $N$  it is possible to approximately measure  $TL1$  for the Dynamite JVM when group block optimisations are disabled,  $TL1$  plus  $TL2$  for the Dynamite JVM with group block optimisation enabled (-O) and  $TL2$  plus  $TL3$  for the HotSpot client and server JVMs.

Table 8.4 and figure 8.5 show the total execution times for Dhrystone when the number of loop unrolls ( $N$ ) is varied for a fixed number of 1,048,576 ( $2^{20}$ ) loop executions. The test machine is the same as in section 8.3.1 and was measured using the time command. Each test was repeated 100 times and the lowest value used. The Dynamite JVM with group block optimisations failed to complete when the loop was unrolled 16 or more times due to a problem in the group block algorithm.

Table 8.5 and figure 8.6 show the time to execute just the inner loop part of the test benchmark. The time is measured using the `java.lang.System.currentTimeMillis` library call.

Tables 8.6 and 8.7 show the increase in translation cost per loop unroll as the loop is unrolled.

The time required to translate a loop of the Dhrystone benchmark by the Dynamite JVM executing without group blocks, does not increase linearly as would be expected from equation 8.4. The number of instructions executed increases in

$N$	Dynamite	Dynamite -O	HotSpot client	HotSpot server
1	29.97	10.43	1.01	1.17
2	33.23	11.08	1.03	1.44
4	37.69	13.72	1.03	2.00
8	43.61	23.60	1.25	3.38
16	105.99		1.67	5.73
32	124.05		2.59	8.03
64	137.31		3.12	12.70

Table 8.4: Translation cost: total execution time

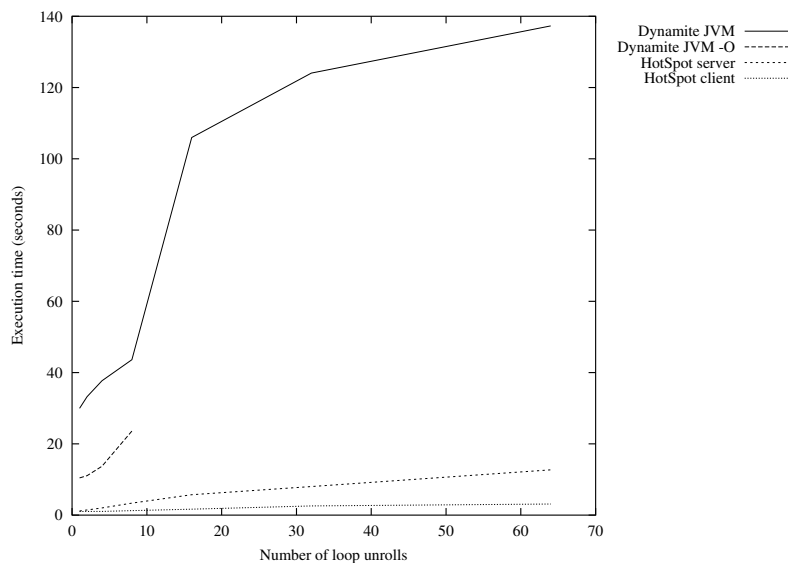


Figure 8.5: Translation cost: total execution time

$N$	Dynamite	Dynamite -O	HotSpot client	HotSpot server
1	29.35	10.40	0.80	1.26
2	34.12	12.42	0.85	2.20
4	37.34	20.67	0.89	3.40
8	42.98	63.39	1.20	6.62
16	114.15		1.73	10.64
32	128.08		2.89	10.76
64	140.61		4.05	24.48

Table 8.5: Translation cost: inner loop execution time

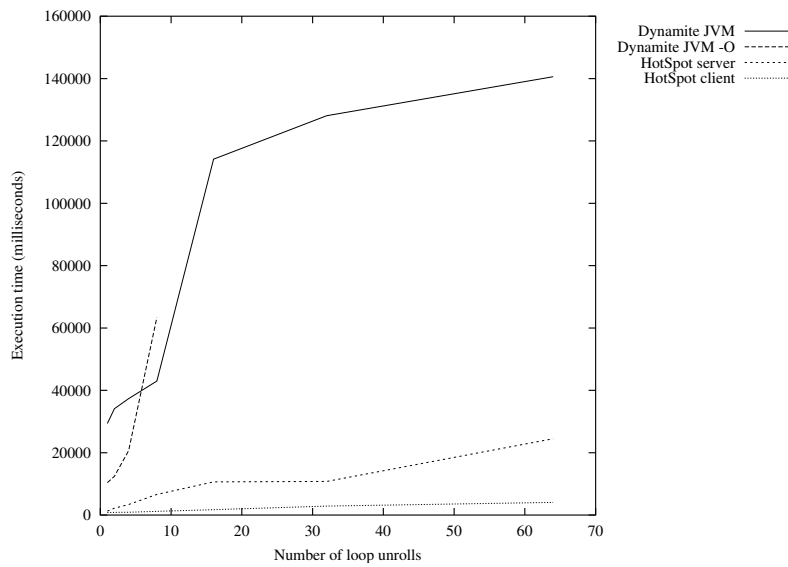


Figure 8.6: Translation cost: inner loop execution time

$N$	Dynamite	Dynamite -O	HotSpot client	HotSpot server
1 to 2	3.26	0.65	0.02	0.27
2 to 4	2.23	1.32	0.00 <sup>2</sup>	0.28
4 to 8	1.48	2.47	0.055	0.345
8 to 16	7.80		0.0525	0.294
16 to 32	1.13		0.0575	0.144
32 to 64	0.42		0.0166	0.146

Table 8.6: Translation cost: increase in total execution time per loop unroll

$N$	Dynamite	Dynamite -O	HotSpot client	HotSpot server
1 to 2	4.77	2.02	0.05	0.94
2 to 4	1.61	4.13	0.02	0.6
4 to 8	1.41	10.68	0.0775	0.805
8 to 16	8.90		0.0663	0.503
16 to 32	0.871		0.0725	0.0075
32 to 64	0.392		0.0363	0.429

Table 8.7: Translation cost: increase in inner loop execution time per loop unroll

accordance to equation 8.4, however, having more instructions translated means that those instructions are located at different addresses. It is probable that this will adversely affect the instruction cache on the test system.

The Dynamite JVM with group block optimisations enabled (-O) again does not have a linear behaviour. The group block translation algorithm orders and optimises basic blocks passing over them several times. The curve of the results in figure 8.6 fits an exponential curve.

In table 8.6, the HotSpot client JVM's results shows it taking 0.00 seconds to translate a copy of the Dhrystone loop from 2 to 4 loop unrolls. This result shows a limit in the resolution of the counter used. It could be a consequence of the HotSpot client using an interpreter to translate bytecodes initially. However, it wouldn't make sense for that large of a loop count to not be optimised by the translator. The other results show the HotSpot client JVM having the lowest average translation overhead. Again the translation overhead does not increase linearly as the Dhrystone loop is unrolled. This can be due to the architecture of the test system or the complexity of the optimisation algorithm. The HotSpot client JVM is designed to have a low latency, which is affected by the translation cost.

The HotSpot server JVM's results show that the HotSpot server JVM is slower to translate a loop of the Dhrystone benchmark than the HotSpot client. The HotSpot server JVM is designed to translate and produce better code than the HotSpot client JVM. This extra performance comes at the cost of translation time and latency.

## 8.5 Back-end performance

The performance of the Dynamite JVM is reliant on its back-end. By examining the target code produced it is possible to identify the potential for optimisation.

Part of the hot path of all the benchmarks is the `String.hashCode` library function. One of its uses is to maintain the `String` packages intern table. The source code, bytecode and Dynamite JVM IR of the Classpath `hashCode` method are shown in figure 5.1. They are repeated for convenience here in figure 8.7.

Figure 8.8 shows the target machine code generated by the Dynamite JVM for

---

<sup>2</sup>A time of 0.00 seconds, to translate a loop of the Dhrystone benchmark, shows the timing counter used to not have enough resolution.

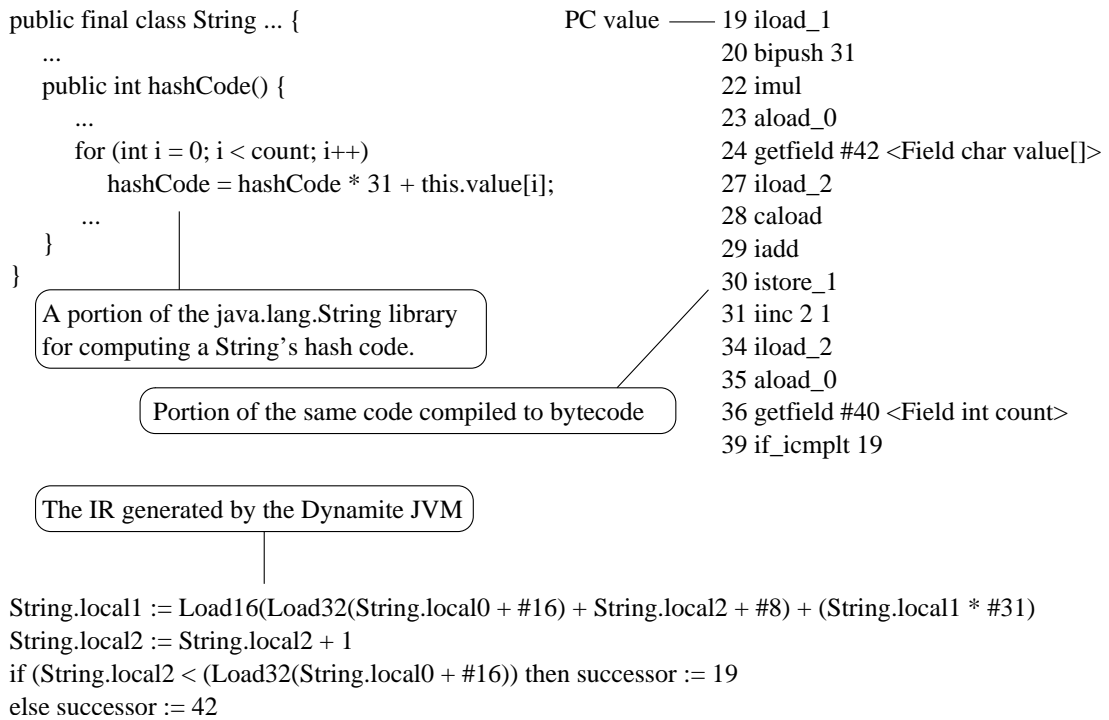


Figure 8.7: Dynamite JVM translation of hashCode method

a group block containing this loop. The IA-32 back-end generated the assembler output, extra comments (lines starting with #) have been added to explain the generated code. The first line of output gives the hexadecimal address of the bytecodes in memory, the hexadecimal address of the IsoBlock and the execution count at the point the group block was created.

The target code has a number of potential inefficiencies in the code generated by the back-end:

- No immediate constants: constant values are generated into registers when they could form part of an instruction. E.g. The target code at instruction 0x4049d725 could be `addl $8,%ebx` thereby eliminating instruction 0x4049d71a and freeing up a register. The constant value can even propagate as far as the load instruction, as loads can be offset by a constant value.
- Read-after-write hazards: the potential for executing instructions in parallel, on architectures that can not re-order instructions, is limited when the value written by an instruction is read by the immediately proceeding instruction. For example, this happens with instructions 0x4049d725 and

```

82075bf (82f13f8) 150
# Register state before entering the block is:
# - ebp contains a pointer to subject machine registers in memory
# - edx contains hashCode.L01
# - esi contains hashCode.L02
# Generate constant 8
0x4049d71a: movl    $0x8,%eax
# Fill ebx with the register value of hashCode.L00
0x4049d71f: movl    0x2c4(%ebp),%ebx
# Generate hashCode.L00 + 8
0x4049d725: addl    %ebx,%eax
# Generate Load (hashCode.L00 + 8)
0x4049d727: movl    (%eax),%eax
# Generate constant 1
0x4049d729: movl    $0x1,%edi
# Spill edx into hashCode.L01
0x4049d72e: movl    %edx,0x2c8(%ebp)
# Copy hashCode.L02 to edx
0x4049d734: movl    %esi,%edx
# Spill esi into hashCode.L02
0x4049d736: movl    %esi,0x2cc(%ebp)
# Generate constant 16
0x4049d73c: movl    $0x10,%esi
# Generate Load(hashCode.L00 + 8) + 16
0x4049d741: addl    %esi,%eax
# Generate hashCode.L00 + 16
0x4049d743: addl    %esi,%ebx
# Generate Load(hashCode.L00 + 16)
0x4049d745: movl    (%ebx),%ebx
# Copy hashCode.L02 to esi
0x4049d747: movl    %edx,%esi
# Swap edi (constant 1) for the value in ecx
0x4049d749: xchgl   %edi,%ecx
# Generate hashCode.L02 << 1
0x4049d74b: shll    %ecx,%esi
# Swap edi and ecx back
0x4049d74d: xchgl   %edi,%ecx
# Generate (Load(hashCode.L00 + 8) + 16) + (hashCode.L02 << 1)
0x4049d74f: addl    %esi,%eax
# Generate Load((Load(hashCode.L00 + 8) + 16) + (hashCode.L02 << 1))
0x4049d751: movswl  0x0(%eax),%eax
# Generate 1 + hashCode.L02
0x4049d758: addl    %edx,%edi
# Generate (1 + hashCode.L02) < Load(hashCode.L00 + 16)
0x4049d75a: cmpl    %ebx,%edi
0x4049d75c: movl    $0x0,%ebx
0x4049d761: setl    %ebx
0x4049d764: andl    $0x1,%ebx
# Fill edx with initial hashCode.L01
0x4049d767: movl    0x2c8(%ebp),%edx
# Generate constant 31
0x4049d76d: movl    $0x1f,%esi
# Generate 31 * hashCode.L01
0x4049d772: imull   %edx,%esi
# Generate constant 65535
0x4049d775: movl    $0xffff,%edx
# Generate Load((Load(hashCode.L00 + 8) + 16) + (hashCode.L02 << 1)) & 65535
0x4049d77a: andl    %edx,%eax
# Generate (Load((Load(hashCode.L00 + 8) + 16) + (hashCode.L02 << 1)) & 65535) + (31 * hashCode.L01)
0x4049d77c: addl    %esi,%eax
# IF (1 + hashCode.L02) >= Load(hashCode.L00 + 16)
0x4049d77e: cmpl    $0x0,%ebx
0x4049d784: je      0x4049d856
# ELSE (1 + hashCode.L02) < Load(hashCode.L00 + 16)
0x4049d78a: movl    %eax,%edx
# Set up esi to match entry register state
0x4049d78c: movl    %edi,%esi
0x4049d78e: jmp     0x4049d71a

```

Figure 8.8: String hashCode method target code

0x4049d727. Section 4.4.1 describes instruction scheduling to improve this.

- Not enough registers: the spill of the initial value of `hashCode.L01` by instruction 0x4049d72e and the subsequent fill by instruction 0x4049d767 would be unnecessary if register pressure had not caused them to be spilled. Register pressure is aggravated by generating constants into registers.
- Inefficient shift instructions: the back-end generated instructions 0x4049d749, 0x4049d74b and 0x4049d74d to perform a left shift of one place. Register `%edi` was a constant so the `shll` instruction could have used an immediate constant. The `shll` instruction could also have been replaced by `leal` or even an `addl` instruction which will typically execute faster.
- Inefficient successor computation: Instructions 0x4049d75a, 0x4049d75c, 0x4049d761, 0x4049d764, 0x4049d77e and 0x4049d784 calculate what the successor to the block is. The first four instructions set the value of register `%ebx`. The next 2 instructions examine the value of `%ebx` to choose the successor. If the value of `Load(hashCode.L00 + 16)` was not destroyed from register `%ebx` by this computation then it could be used itself at in the `cmpl` instruction 0x4049d764 with the value that is still in register `%edi`. This would make instructions 0x4049d75a, 0x4049d75c, 0x4049d761 and 0x4049d764 redundant.
- Cold path tested first: The successor test computes whether it should branch to the cold path before it falls through to take the hot path. At least one instruction can be saved from the hot path by testing the hot path condition first. This is likely to improve branch prediction on the target processor.

A more optimal hand coded version of the code is shown in figure 8.9, the instruction addresses shown are related to the original loop and would be different if the code were compiled.

The more optimal loop reduces the number of instructions executed per loop from 34 instructions to 19. This does not necessarily mean that performance will be increased on a target processor due to hardware optimisation.

```

82075bf (82f13f8) 150
# Register state before entering the block is:
# - ebp contains a pointer to subject machine registers in memory
# - edx contains hashCode.L01
# - esi contains hashCode.L02
# Fill ebx with the register value of hashCode.L00
0x4049d71f: movl    0x2c4(%ebp),%ebx
# Generate Load (hashCode.L00 + 8)
0x4049d727: movl    8(%ebx),%eax
# Spill edx into hashCode.L01
0x4049d72e: movl    %edx,0x2c8(%ebp)
# Copy hashCode.L02 to edx
0x4049d734: movl    %esi,%edx
# Spill esi into hashCode.L02
0x4049d736: movl    %esi,0x2cc(%ebp)
# Generate Load(hashCode.L00 + 8) + 16
0x4049d741: addl    $16,%eax
# Generate Load(hashCode.L00 + 16)
0x4049d745: movl    16(%ebx),%ebx
# Copy hashCode.L02 to esi
0x4049d747: movl    %edx,%esi
# Generate hashCode.L02 << 1
0x4049d74b: addl    %esi,%esi
# Generate (Load(hashCode.L00 + 8) + 16) + (hashCode.L02 << 1)
0x4049d74f: addl    %esi,%eax
# Generate Load((Load(hashCode.L00 + 8) + 16) + (hashCode.L02 << 1))
0x4049d751: movswl  0x0(%eax),%eax
# Generate 1 + hashCode.L02
0x4049d758: leal    1(%edx),%edi
# Fill edx with initial hashCode.L01
0x4049d767: movl    0x2c8(%ebp),%edx
# Generate 31 * hashCode.L01
0x4049d772: imull   $31,%edx
# Generate Load((Load(hashCode.L00 + 8) + 16) + (hashCode.L02 << 1)) & 65535
0x4049d77a: andl    $65535,%eax
# Generate (Load((Load(hashCode.L00 + 8) + 16) +
#           (hashCode.L02 << 1)) & 65535) + (31 * hashCode.L01)
0x4049d77c: addl    %esi,%edx
# Set up esi to match entry register state
0x4049d78c: movl    %edi,%esi
# IF (1 + hashCode.L02) < Load(hashCode.L00 + 16)
0x4049d77e: cmpl    %ebx,%edi
0x4049d784: jl      0x4049d71a
# ELSE (1 + hashCode.L02) >= Load(hashCode.L00 + 16)
0x4049d78e: jmp     0x4049d856

```

Figure 8.9: Optimised string hashCode method target code

## 8.6 Summary

Chapters 6 and 7 described the implementation and optimisations of the Dynamite JVM. The implementation decisions were measured and the lowest overhead scheme chosen. This chapter has performed measurements of more general aspects of JVMs. In two of the sections, the Dynamite JVM has been examined and compared to the HotSpot client and server JVMs using a number of tests based around a Java version of the Dhrystone benchmark.

Section 8.4 shows that translation cost does not increase linearly for the Dynamite JVM. The translation cost is much larger for the Dynamite JVM than the HotSpot client and server JVM. This impacts on the latency of the Dynamite JVM. The translation cost should improve in comparison to a method based JIT compiler as the Dynamite JVM has to translate fewer bytecodes, as shown in section 8.2.

Section 8.3 shows the performance of the translated code of Dynamite JVM. Again the performance is not as good as the HotSpot client and server JVM. The performance is 4.16 times slower than the HotSpot client JVM and 4.98 times slower than the HotSpot server JVM per execution of the Dhrystone loop. The relative performance is between 3 and 4 times worse when measured by an external timing program.

Section 8.5 shows an example of a group block translation by the Dynamite JVM. The code is sub-optimal and on the considered piece of frequently executed code the instruction count can be reduced by more than 40%.

This thesis ends with a description and conclusion of the work presented in the previous chapters. It also describes future work to improve the Dynamite JVM, in terms of completeness and performance.

# Chapter 9

## Summary and Conclusions

This thesis has introduced an implementation of a JVM called the Dynamite JVM. In section 9.1 the work from the previous chapters is summarised. Section 9.2 draws conclusions from this work. Finally, section 9.3 considers future work and optimisation to the Dynamite JVM.

### 9.1 Summary

#### 9.1.1 Background

Chapter 2 introduced the Java programming language, the Java Virtual Machine and the class libraries. Key features of Java to the JVM are object representation, the class file format and dynamic link loading. Interpreting, hardware and dynamic compiling JVMs were described; the HotSpot JVM was described as an example of a state-of-the-art JVM.

Chapter 3 discussed existing dynamic binary translator environments providing context for the Dynamite DBT, and introducing techniques used by DBTs. Chapter 4 provides a description of Dynamite, the framework on which the Dynamite JVM is built. DBTs are significantly different to JVMs, there are certain design issues and opportunities unique to making a JVM from a DBT.

#### 9.1.2 Design

Chapter 5 described key features of the Dynamite JVM, such as how it provides exception support, loads classes and performs native method call. The expression stack was introduced as a mechanism for mapping the JVMs stack on to registers

and eliminating its associated overhead. Threading and array exceptions were described, with this support the Dynamite JVM would be 100% Java compatible.

Chapter 6 presented two schemes that allow the Dynamite JVM to optimise across method boundaries by mapping a method's stack frame to a fixed set of registers. With the chosen fixed register window scheme there is a parameter passing penalty, however, this scheme does avoid the need for multiple re-translations. The overhead can be eliminated by group block formation.

Chapter 7 presented the problem and solutions to recursion detection and fix-up for the fixed register-window optimisation. Dynamic loading of classes means full program analysis is not possible at runtime, so the Dynamite JVM uses lazy recursion detection. This technique requires stack fix-up to be performed. Immediate and delayed fix-up techniques were described, the immediate stack fix-up scheme is used in the Dynamite JVM.

### 9.1.3 Experiments

The decision of which inter-procedural optimisation technique to use was made in chapter 6 from analysis of a set of benchmarks. For the fixed register-window scheme to work, fix-up blocks are required at the boundary of virtual methods to perform parameter passing. In the worst case, less than 1.3% of the dynamic instruction mix is made of fix-up instructions (which are effectively eliminated during group block creation). The alternate sliding register-window scheme was found to require between 1.819 and 5.103 extra translations per method.

Analysis was performed on the fixed register-window scheme to determine how many registers would be used by the Dynamite JVM as larger group blocks were formed. Ignoring reuse of registers, over 80% of a program's execution can fit into just 100 registers.

A stack in memory is used when recursion is detected. Chapter 7 showed an example of stack fix-up, an operation performed when recursion is detected. Recursive method calls were found to make up to 0.16% of the dynamic instruction mix. In the worst case 9.576% of executed virtual method calls were found to be recursive. Simple liveness analysis was performed on the frames to be placed on the memory stack. It was found that between 3 and 9 words of data had to be placed on the memory stack per method call. Stack fix-up was performed between 4 and 35 times on the benchmark programs, and in the worst case 4538 words of data would need re-ordering.

Chapter 7 measured the performance of the Dynamite JVM on the highly recursive Takeuchi benchmark, compared to the HotSpot client and server JVMs. Performance was found to be 3.163 times slower per executed bytecode than the HotSpot client JVM, and 5.097 slower compared to the HotSpot server JVM. Two regions of dead code were highlighted in the hot region of the IR graph which could be eliminated to improve performance.

Chapter 8 measured the translation saving afforded by basic block rather than method based translation. Up to 18% of the bytecodes translated by a method based translator were found to be never translated or executed by the Dynamite JVM. The execution performance of the Dynamite JVM was found to be 4.16 times slower than the HotSpot client JVM and 4.98 times slower than the HotSpot server JVM per execution of the Dhrystone loop. An example of the code produced by the Dynamite JVM was examined and 40% of the produced instructions could be removed by using existing optimisations within Dynamite.

The Dynamite JVM's translation cost was found to be significantly worse than the HotSpot JVMs. For the benchmark in chapter 8, when user time was measured, instead of the time in the inner loop, the performance was also significantly worse. The large memory footprint and the lack of an interpreter are likely to be hurting the Dynamite JVM's performance. Section 9.3 describes improvements to these aspects of the Dynamite JVM.

## 9.2 Conclusions

Existing dynamic optimisation techniques have been applied from a dynamic binary translator and used to create a JVM environment. This has not been at the expense of throwing away the rich amount of extra information Java gives its dynamic execution environment. Although the techniques described have been demonstrated to have low overhead, the thesis hasn't shown they result in a JVM faster than the state-of-the-art. Consideration on how this performance can be improved has been presented throughout the thesis. Improving the translation of the IR to target machine code, and optimising the class library are likely to improve performance the most. Other techniques are described in the next section.

Specific contributions of this work have been:

- The design, implementation and demonstration of dynamically compiling

JVM potentially with full support for all Java applications. In common with other JVMs, the Dynamite JVM maps the Java stack onto registers to eliminate the overhead of bytecodes that manipulate the stack.

- The application of basic block translation techniques which support trace scheduling to Java bytecode. This is novel as all previous dynamic compiling Java environments orient themselves on compiling methods<sup>1</sup> and translation of basic blocks can result in as much as 18% fewer bytecodes being translated.
- The generation of a register allocation algorithm that allows a basic block optimisation algorithm to perform optimisations over method boundaries, as with method inlining. This scheme is applicable to over 90% of all methods.
- The generation of a low cost lazy recursion detection scheme that can be used in dynamic binary translators to map more of the stack into the register stack cache. A cheap fix-up scheme is described and implemented to catch cases where recursion is detected late.
- The measurement of the performance of JVM environments to calculate the cost of translation and the quality of the translated code.

The JVM shares similarities with a number of virtual machines and computer architectures. By recreating high-level procedure call and return semantics within the dynamic binary translator, fixed register windows and lazy recursion detection can be used with other architectures and virtual machines.

### 9.3 Future work

The Dynamite JVM does not currently support certain features required to run all Java programs. Sections 9.3.1, 9.3.2 and 9.3.3 consider the implementation of these features. Section 9.3.4 considers the alternative class library implementations for the Dynamite JVM, as this is a performance concern from chapter 8. Section 9.3.5 considers improvements to the dynamic memory footprint of the

---

<sup>1</sup>Other JVMs perform partial translation of methods such as Latte [LYK<sup>+</sup>99]. Other JVMs such as Jalapeno [AAea00] are able to reschedule basic blocks in the presence of JVM features such as exceptions.

Dynamite JVM. Sections 9.3.6 and 9.3.7 consider how latency, and the overall performance of the dynamic code produced by the Dynamite JVM, can be improved.

### 9.3.1 Threading

By implementing the `java.lang.Thread` set of native functions and by allowing multiple call stacks, threading support is added to a JVM. In a conventional JVM these call stacks are in memory: in the Dynamite JVM the call stack is split between memory, for recursive methods, and an abstract register pool (as described in chapter 6). The address of the abstract register pool is held in a register. By allocating multiple abstract register pools and switching the value held within the register pointing to the abstract register pool, different thread contexts can be supported. This is an extension of the current mechanism for running class initialisers and has the advantage that no methods need re-translating.

An alternative implementation would be to allocate the abstract register pools at known addresses in memory and specialise translated code to the abstract register pool. This would free the register holding the address of the abstract register pool, and having this extra register available would reduce the number of spills and fills from the abstract register pool. However, as the code is specialised to expect the abstract register pool to appear at a fixed address, each thread will need a separate translation specialised to the address of the abstract register pool. Re-translation of separate threads can be avoided on processors with virtual memory, by altering the page table on a thread context switch so that the virtual addresses always map to the appropriate abstract register pool real addresses for a particular thread context.

### 9.3.2 Garbage collection

Section 5.4.2 introduced the Dynamite JVM implementation of the new bytecode for allocating memory. Garbage collection is performed by a JVM to reclaim allocated memory. The Dynamite JVM has the ability to perform exact garbage collection on basic block boundaries as the types of values in abstract registers are held for each basic block boundary. For example, mark and compaction garbage collection can be performed by examining the abstract register pool on a basic block boundary and generating a root set of object references. By then examining

the objects pointed to by these references, further objects can be found until all reachable objects have been discovered (Dynamite JVM object layout is described in section 5.3). After all reachable objects are marked, all remaining objects can then be freed from memory.

This scheme is sub-optimal as it does not allow easy parallelisation and it requires execution to be suspended whilst it is performed. For a description of many of the different ways of implementing a garbage collector, Wilson [Wil92] provides a survey considering implementations for uniprocessor systems, whilst Flood et al. [FDSZ01] consider a parallel implementation of a garbage collector.

Escape analysis is a technique for discovering whether an object is referenced outside of a loop, method or by another executing thread [Bla99]. By knowing an object doesn't escape a loop, method or thread, the object may be allocated on the stack or reused around a loop body. By reducing the number of allocations through reuse and using the stack: less time is spent allocating objects, less time is spent in the garbage collector and the locality of objects is improved. Profile directed escape analysis [VR01] would be suited to the Dynamite JVM within group block creation.

### 9.3.3 Exception handling

Section 5.2 introduced how the Dynamite JVM supports runtime exceptions and techniques for improving their performance. The following solutions were identified:

- stack cutting - Section 2.6 introduced how holding a stack of catch addresses, and the associated exceptions, for all try regions that have been entered, can allow for more efficient exception dispatch. However, there is an associated cost of maintaining a runtime data structure as the bytecodes enter and leave try regions.
- bound load store IR - the Dynamite JVM's IR is currently able to eliminate unnecessary array boundary checks through constant propagation and, in the future, using value specific optimisation. Array boundary checks do, however, force basic blocks to end early. If bounded loads and stores were an IR operation, they could be translated and scheduled within a basic block without the need to terminate the block.

- dummy null object - To speed dispatch of exceptions caused by virtual calls on null object references, a special virtual method table for the null object will re-direct dynamic code into a null exception handler.

By examining bytecodes further, more potential for optimisation can be identified. For example, precise exceptions are not necessary if a method accesses an array and does not have any catch regions for array bound exceptions.

The benchmarks examined in chapters 6 and 8 did not exhibit worst case exception behaviour and, as such, optimisation of the exception mechanism would not have greatly improved the performance of the Dynamite JVM. The `_202_Jess` benchmark uses the `athrow` bytecode to generate exceptions to signal the recognition of parsed tokens; these exceptions are caught by a method several stack frames below the parser. `_202_Jess` would benefit from exception prediction (described in section 2.6) to avoid throwing the same exception multiple times at different stack depths until the handling method and frame are found.

### 9.3.4 Class library

The Dynamite JVM uses the Classpath [GNU02] class library implementation. Benchmarks such as `_213_javac` from the SpecJVM 98 [SPE98] benchmark suite were unable to run because the class library implementation was incompatible. Improvements in the class library will allow the Dynamite JVM to run a wider range of benchmarks.

The Dynamite JVM lacks full support for features required by the class library. Threading and loading classes into the JVM require special JVM classes which are only partially present. Full support for these features will allow the Dynamite JVM to run all Java applications.

Section 8.3 looked at the performance of the Dynamite JVM in comparison with the HotSpot server and client JVM at executing the Dhrystone benchmark. The Dynamite JVM was between 4.16 and 4.98 times slower than the HotSpot client and server JVM. In section 7.3.3 an analysis of the Takeuchi function showed the performance of the Dynamite JVM to be 3.16 times slower than the HotSpot client JVM and 5.10 times slower than the HotSpot server JVM. This performance was measured with an external timer, when the same timer was used on the Dhrystone benchmark performance was 3 to 4 times slower. A key reason why the Dhrystone benchmarks appears slower than the Takeuchi function is that

the Takeuchi function makes no use of the Java class library, whereas Dhrystone uses it for string access and comparison.

Calls to the class library can be replaced with pre-optimised code for certain known library functions; however, this requires a tight integration between the JVM and the class library. This tight integration can be at the expense of the portability of the JVM between different class library implementations. The class library's performance can be further improved by light weight access to the Java data structures by native methods, such as with KNI or CNI, as described in section 2.4. The Classpath class library uses the JNI native interface, which requires an indirect method call to perform object and JVM manipulation from the native code. The CNI interface can be used, but this restricts the object layout.

### 9.3.5 Memory usage

JVM interpreters and hardware are able to execute Java bytecodes without a region of memory for storing translated bytecodes. For certain embedded applications, having a small memory footprint can be as important as execution performance.

The Dynamite JVM's fixed register window scheme currently requires a pool of 8192 registers which occupies 32 kilobytes of memory for each thread within the JVM. The number of registers can be reduced by sharing register windows of methods that can not execute at the same time, such as leaf methods. The number of used registers can also be reduced by releasing register windows for methods that are infrequently accessed or only accessed at start-up. This would require re-translation of the method if it were ever accessed again. Alternatively, all methods could use one frame, and the memory stack and fixed register windows can be used for methods that are re-translated for group blocks.

The Dynamite JVM currently performs bytecode verification at translation time. This is a violation of the JVM specification [Sun95] and it also requires extra state to be held in basic blocks so that bytecodes may be verified when they are translated. A bytecode verifier in the class loader would remove the need for this state information and free memory.

Certain target code is only executed once at start-up; however, currently this target code is never destroyed. By holding a time-stamp of when a basic block or group block was last executed, it is possible to work out which blocks

were executed recently and, assuming temporal locality of code, free all other translated blocks from memory.

### 9.3.6 Latency

Section 8.3 showed how the time taken to execute a fixed number of iterations of the Dhrystone loop varied over time between the Dynamite JVM, with and without group block optimisation, and the HotSpot client and server JVMs. By not performing complicated translation, as with the HotSpot server and Dynamite JVM with group block optimisation, latency is improved for shorter runs. Performing little translation is therefore an advantage for short runs. The Dynamite JVM in basic block mode has to translate single blocks of code, which is slower than just interpreting the bytecodes. Subsequently, a bytecode interpreter would improve the Dynamite JVM's latency performance for short runs.

The Dynamite JVM performs large group block optimisation and register window optimisation shortly into the execution of a benchmark. By phasing in the optimisation it would again be possible to improve latency. Aggregate group blocks are one solution to this problem. An aggregate group block is a group block where translation has been terminated early so that the group block contains fewer members than a normal group block. By limiting the translation, the translation time is reduced, and fix-up of branches between aggregate group blocks allows for similar performance to the normal group block (synchronising register maps results in lower performance).

Another optimisation that can be phased in is the inter-procedural optimisations described in chapters 6 and 7. If the normal case used a memory stack and the optimal case used the fixed register window scheme, there would be no translation of fix-up code except for hot-regions. This could decrease translation cost (and hopefully latency), but the complexity of having two translations schemes would complicate translation of method calls and thereby potentially increase the translation cost. Recursion detection is necessary even if fixed-register windows are only used in a more optimised case.

### 9.3.7 Performance

Section 8.5 showed how the Dynamite JVM can produce better code than it currently does. Chapter 4 described constant propagation, code motion, code duplication, alias optimisation, value specific optimisation, instruction scheduling and idiom recognition optimisations within the Dynamite JVM. The Dynamite JVM only supports a limited amount of constant propagation, instruction scheduling and idiom recognition. Further to the optimisations discussed in chapter 4, the Dynamite JVM would also benefit from call IR. Without nodes representing subroutine or method call and return within the Dynamite JVM means that constant propagation of return addresses can not be performed. This is necessary to stop a computed jump being used for subroutine return (as described in chapter 6). Method inlining eliminates this computed jump in Java JIT compilers.

Another optimisation that may later be of use within the Dynamite JVM is speculative thread-level parallelisation. Dynamic translation allows a number of run-time optimisations to be performed, many of which may prove to be unsafe at some point. Speculative threading takes advantage of this by running the heavily optimised code on a separate thread and then running the correctness checks and potentially safer translations on other threads. Speculative threading can also parallelise code, having multiple translations running at the same time. When the code has been run and checked, the state changed can be committed to memory and the register bank, in order to not violate the sequential semantics of the program. Speculative threading is a good optimisation on SMT architectures where some processor cores are idle. Speculative threading also benefits from hardware support for incorrect speculation that allows quick roll-back (see section 3.4.1).

## 9.4 Final remark

Basic block dynamic compilation is a viable method for executing and optimising Java programs. It reduces the amount of translated and generated code. Having comparable performance with a state-of-the-art JVM has been beyond the scope of this thesis due to the heavy optimisation of such a JVM. However, a significant contribution has been made linking dynamic binary translator technology and optimisations with that of JVMs.

# Appendix A

## feJAVA Dynamite JVM Front-End

This appendix describes the front-end source files of the Dynamite JVM project including some more of the low-level implementation.

- `Class.cc` / `Class.h` - These files contain the implementation of the `Class` object which is used to represent all Dynamite JVM classes. Methods on the `Class` object allow for class loading, object creation and accessor methods to Dynamite JVM objects.
- `ExceptionStack.cc` / `ExceptionStack.h` - These files contain the `ExceptionStack` class which is used to provide run-time support for the stack cutting technique of Java exception handling (see section 2.6).
- `FrameDescriptor.cc` / `FrameDescriptor.h` - These files contain the `FrameDescriptor` class which is used to describe a method's frame within the Dynamite JVM. The `FrameDescriptor` class handles allocation of abstract registers to the Dynamite JVM and provides accessor methods to the relevant part of the frame and support for the memory stack.
- `FrontEnd.cc` / `FrontEnd.h` - The `FrontEnd` class provides the interface between the Dynamite JVM and Dynamite. It is responsible for creation and initialisation of all front-end objects, defining basic blocks for the Dynamite kernel and clean-up after the Dynamite JVM has finished execution.
- `Fuse.cc` / `Fuse.h` - The `Fuse` class is provided to support kernel debugging features of Dynamite.

- GC.cc / GC.h - The GC class is responsible for garbage collection although the collect method is currently unimplemented. All constructed objects are registered with the GC object and it provides debug support.
- Instruction.cc / Instruction.h - The instruction object provides functions for classifying and debugging bytecodes.
- IsoBlock.cc / IsoBlock.h - Defines the IsoBlock class (an IsoBlock is a combination of a basic block with compatibility information). This class contains the methods to translate a block, defining compatibility and the expression stack.
- JMethodID.cc / JMethodID.h - The method object holds information on translated methods and the support routines for lazy recursion detection.
- MethodArea.cc / MethodArea.h - The method area is the pool of loaded classes and methods. Separate class loaders have separate method areas.
- NullObject.cc / NullObject.h - Definition of fast null pointer class.
- SubCall.cc / SubCall.h - Substitute calls for new, instanceof and expressions not supported by the Dynamite kernel.
- SubCallInvokes.cc - Substitute calls for performing JNI native method call.
- jni.cc - JNI support code that allows native methods to interrogate objects and the JVM, as well as allowing native methods to call java methods.
- native.cc - the Classpath native library does not provide all of the native methods that need implementing for the class library to function as some of the native methods are JVM dependent. These native methods are defined here.
- ByteCode.h - list of bytecodes including Dynamite JVM \_quick bytecodes.
- Types.h - definition of types used within the JVM.
- build.pl / InstructionData - script to generate expression or substitute call planting code dependent on data file definition.
- registerID.h - JVM register definitions used by the Dynamite kernel and for debug support.

# Appendix B

## Result Data

This appendix contains the data used for the graphs and tables in chapters 6, 7 and 8.

### **B.1 Number of fix-up blocks for fixed register-window scheme**

Section 6.4.2 measured the number of translated and executed method call bytecodes in a fixed register-window scheme. The data used to calculate the percentage of executed bytecodes in table 6.2 are shown in table B.1

### **B.2 Number of retranslations for sliding register-window scheme**

Section 6.4.3 measured the number of translations necessary for the sliding register-window scheme. Tables B.2 and B.3 show the data presented in figure 6.5. The results for `_202.jess` are presented in a separate table due to their length. In certain cases when the number of methods retranslated was zero the row of the table has been omitted for brevity. In each table translations is the number of translations of that method performed, methods is the number of methods translated that number of times.

helloworld	Translated	Executed	mpeg2dec	Translated	Executed
invokestatic	41	94	invokestatic	441	3383318
invokespecial	68	192	invokespecial	1037	3711
invokevirtual	89	281	invokevirtual	168	9045
invokeinterface	0	0	invokeinterface	0	0
fixup_iv	152	471	fixup_iv	1199	12755
total	4398	21877	total	27172	604548289

mpeg2enc	Translated	Executed	_201_compress	Translated	Executed
invokestatic	352	5414887	invokestatic	252	12652
invokespecial	812	6169	invokespecial	922	2423450
invokevirtual	184	4221232	invokevirtual	520	15775531
invokeinterface	0	0	invokeinterface	10	50
fixup_iv	988	4227400	fixup_iv	1412	18199019
total	33437	14693689649	total	19570	1212483527

_202_jess	Translated	Executed
invokestatic	496	126818
invokespecial	1264	262784
invokevirtual	1493	5731643
invokeinterface	26	6532
fixup_iv	2677	6000943
total	33907	148612987

Table B.1: Number of method call bytecodes translated and executed in fixed register-window scheme

helloworld		mpeg2enc	
Translations	Methods	Translations	Methods
1	34	1	115
2	16	2	33
3	8	3	17
4	3	4	9
5	2	5	5
6	1	6	7
7	1	7	5
8	0	8	3
		9	1
		10	0
mpeg2dec		_201_compress	
Translations	Methods	Translations	Methods
1	81	1	71
2	36	2	66
3	22	3	24
4	13	4	28
5	9	5	10
6	5	6	5
7	5	7	5
8	7	8	8
9	2	9	3
10	1	10	3
11	1	11	4
12	0	12	2
13	1	13	4
		16	1
		17	2
		23	1

Table B.2: Number of translations necessary for methods using the sliding register-window scheme

<u>_202_jess</u>					
Translations	Methods	Translations	Methods	Translations	Methods
1	88	15	3	29	1
2	117	16	11	30	2
3	41	17	3	31	2
4	26	18	5	35	1
5	17	19	3	36	1
6	11	20	1	37	1
7	10	21	3	38	1
8	15	22	1	39	2
9	9	23	2	42	1
10	5	24	3	43	1
11	5	25	6	53	2
12	6	26	6	59	1
13	3	27	1	73	1
14	3	28	3		

Table B.3: Number of translations necessary for methods using the sliding register-window scheme (\_202\_jess)

### B.3 Input to Takeuchi function

The input to the Takeuchi function used in section 7.3.3 is shown in table B.4. The number of bytecodes was determined using the function shown in figure B.1 that assumes 5 bytecodes for the execution of the main function, 23 bytecodes for the false route through the Takeuchi benchmark and 5 bytecodes for the true route. These values were determined by disassembling the benchmark with the javap tool [Sun02b]. The results were sorted and input values to the Takeuchi function chosen for executing a unique number of bytecodes and the number of bytecodes lying between a particular range.

X	Y	Z	bytecodes	X	Y	Z	bytecodes	X	Y	Z	bytecodes
6	0	3	352	2	0	77	226566	5	0	40	78595334
1	0	12	846	3	0	27	266808	4	0	75	106650126
1	0	20	1454	2	0	92	323162	4	0	80	137558832
1	0	28	2062	6	0	11	359908	4	0	85	174739856
3	0	6	2784	3	0	34	526804	6	0	34	228694992
1	0	47	3506	3	0	39	790144	5	0	52	280463874
1	0	56	4190	3	0	45	1206738	6	0	37	371149392
1	0	62	4646	3	0	50	1648868	6	0	40	580634994
1	0	72	5406	5	0	19	2211268	6	0	43	880235012
1	0	82	6166	4	0	30	2951432	5	0	71	1281618410
1	0	92	6926	7	0	14	3759692	7	0	36	1865463140
2	0	15	8636	4	0	34	4802602	5	0	82	2595826750
3	0	10	14108	4	0	36	6000438	5	0	87	3470939922
2	0	26	25964	3	0	83	7429656	5	0	92	4567465642
2	0	32	39302	7	0	16	9118756	7	0	43	6069941676
2	0	38	55376	3	0	95	11106688	6	0	65	9640314934
2	0	44	74186	5	0	28	14115946	7	0	50	16620180584
5	0	10	98278	4	0	49	20017346	7	0	54	27843337856
2	0	57	124346	5	0	33	31077588	6	0	84	43122767242
4	0	14	153872	4	0	59	41480886	6	0	90	64621544604
3	0	24	188604	4	0	65	60703946				

Table B.4: Takeuchi input

```

class takeuchi
{
    public static int result;
    public static long bytecodes;
    public static int tak(int x, int y, int z) {
        if (y >= x){
            bytecodes += 5;
            return z;
        }
        else {
            bytecodes += 23;
            return tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y));
        }
    }
    public static void main(String args[]) {
        int stop = 100;
        for (x=0; x < stop; x++){
            for (y=0; y < stop; y++){
                for (z=0; z < stop; z++){
                    bytecodes = 5;
                    result = tak(x,y,z);
                    System.out.println (''x '' + x + ''y '' + y + ''z '' + z +
                                         ''bytecodes '' + bytecodes + ''\n'');
                }
            }
        }
    }
}

```

Figure B.1: Function to calculate the number of bytecodes executed by the Takeuchi function

# Bibliography

- [AAea00] B. Alpern, C. Attanasio, and et al. The Jalapeno virtual machine. *The IBM Systems Journal*, 39(1), 2000.
- [AD00] Ole Agesen and David Detlefs. Mixed-mode bytecode execution. Technical Report SMLI TR-2000-87, Sun Microsystems Inc., June 2000.
- [AKS00] Erik Altman, David Kaeli, and Yaron Sheffer. Welcome to the opportunities of binary translation. *IEEE Computer*, 33(3):40–45, March 2000.
- [AP98] Denis N. Antonioli and Markus Pilz. Analysis of the Java class file format. ifi-98.04, Department of Computer Science, University of Zurich, April 28 1998.
- [App94] Macintosh Application Environment 2.0. Technical report, Apple Computer Inc., Cupertino, California, USA, 1994.
- [App96] Apple enhances Macintosh Application Environment for HP and Sun workstations. <http://product.info.apple.com/pr/press.releases/1997/q1/961118.pr.rel.mae.html>, November 1996.
- [AS92] Kristy Andrews and Duane Sand. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 213–222, New York, NY, 1992. ACM Press.
- [ASU86] Alfred V. Aho, R. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

- [BDB99] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical report, HP Laboratories Cambridge, 1999.
- [BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.
- [Bed90] R. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 53–64, Berkeley, CA, 1990. USENIX Association.
- [BKMM87] Arndt B. Bergh, Keith Keilman, Daniel J. Magenheimer, and James A. Miller. HP 3000 emulation on HP precision architecture computers. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 38(11):87–89, 1987.
- [BKP99] Zoran Budimlic, Ken Kennedy, and Jeff Piper. The cost of being Object-Oriented: A preliminary study. *Scientific Computing*, 7(2):87–95, 1999.
- [Bla99] Bruno Blanchet. Escape analysis for Object-Oriented languages: application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.
- [BNWK03] Alex Brown, Geraint North, Frank Thomas Weigel, and Gareth Anthony Knight. Method and apparatus for performing native binding. GB Patent Number GB2404042, September 2003.
- [Boc01] bochs: the cross platform IA-32 emulator. <http://bochs.sourceforge.net/>, 2001.
- [BVG97] Aart J. C. Bik, Juan E. Villacis, and Dennis B. Gannon. javar: A prototype Java restructuring compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, 1997.
- [Cal02] Caldera Wabi for Linux user’s guide. <http://www.caldera.com/support/docs/wabi/>, 2002.

- [CE99] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *Proceedings of the International Workshop on Program Comprehension*, pages 192–199. IEEE Computer Society Press, May 1999.
- [CE00] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, March 2000.
- [CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Joint International Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.
- [CKK<sup>+</sup>95] Igor Chebotko, Peter Kalatchin, Yuri Kiselev, Kiril Malakhov, Yuri Petrenko, Efim Podvoisky, Mike Schmit, Sergei Shkredov, Gennady Soudlenkov, and Dan Wroski. *Assembly Language Master Class*. Wrox Press Ltd., 1995.
- [CLCG00] W. Chen, S. Lerner, R. Chaiken, and D. Gilles. Mojo: A dynamic optimization system. In *In Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.
- [Com02] Compaq. *FreePort Express User's Guide*. Compaq, 2002.
- [Cor00] David Cormie. Jazelle - ARM architecture extensions for Java applications, November 2000.
- [Cor05] Transitive Corporation. Technology overview. <http://www.transitive.com/technology.htm>, October 2005.
- [Cra01] Cray T3E 1350. Technical report, Cray Inc., 2001.
- [CS95] Bryce Cogswell and Zary Segall. Timing insensitive binary to binary translation of real time systems. Technical report, Carnegie Mellon University, ECE Dept., Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, 1995.
- [CS00] Cristina Cifuentes and Doug Simon. Procedural abstraction recovery from binary code. In *Proceedings of the European Conference on*

- Software Maintenance and Reengineering*. IEEE Computer Society Press, March 2000.
- [CSB96] J. Bradley Chen, Michael D. Smith, and Brian N. Bershad. Morph: A framework for platform-specific optimization. White Paper TR-0496, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, March 1996.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
- [Dic01] David Dice. Implementing fast Java monitors with relaxed-locks. In *USENIX Java Virtual Machine Research and Technology Symposium*. Sun Microsystems Inc., USENIX, April 2001.
- [EA97] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 26–37, New York, June 2–4 1997. ACM Press.
- [EAGS00] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic translation and optimization. IBM research report, IBM Research Division, July 2000.
- [EAH97] Kemal Ebcioglu, Erik R. Altman, and Erdem Hokenek. A Java ILP machine based on fast dynamic compilation. In *IEEE Mascots International Workshop on Security and Efficiency Aspects of Java*. IEEE Computer Society Press, January 1997.
- [EASG99] Kemal Ebcioglu, Erik R. Altman, Sumedh Sathaye, and Michael Gschwind. Execution based scheduling for VLIW architectures. *Lecture Notes in Computer Science*, 1685:1269–1280, 1999.
- [EM01] Ahmed H.M.R. El-Mahdy. *A Vector Architecture for Multimedia Java Applications*. PhD thesis, The University of Manchester, 2001.

- [Eng96] Dawson R. Engler. VCODE : A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 160–170, New York, May 21–24 1996. ACM Press.
- [FDSZ01] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, 2001.
- [Fla01] Nick Flaherty. Figure fight. *EETimes*, UK, July 2001.
- [Fou02a] The Free Software Foundation. The Cygnus Native Interface for C++/Java integration. <http://sourceware.cygnus.com/java/papers/cni/t1.html>, 2002.
- [Fou02b] The Free Software Foundation. The GCJ home page. <http://sourceware.cygnus.com/java/gcj.html>, 2002.
- [Fou02c] The Free Software Foundation. GNU Compiler Collection. <http://gcc.gnu.org/>, 2002.
- [GCH00] Manish Gupta, Jong-Deok Choi, and Michael Hind. Optimizing Java programs in the presence of exceptions. In *ECOOP*, pages 422–446, 2000.
- [GH01] E. Gagnon and L. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *USENIX Java Virtual Machine Research and Technology Symposium*. USENIX, April 2001.
- [GJS00] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, second edition, 2000.
- [GM88] John C. Goettelmann and Christopher J. Macey. Method and apparatus for direct conversion of programs in object code form between different hardware architecture computer systems. US Patent No. 5,313,614, May 1988.
- [GM95] James Gosling and Henry McGilton. *The Java Language Environment: a white paper*. Sun Microsystems, October 1995.

- [GNU02] GNU. GNU Classpath. <http://www.classpath.org/>, 2002.
- [Gre98] Anthony Green. libffi - foreign function interface. <http://sources.redhat.com/libffi/>, 1998.
- [Gri99] Robert Griesemer. Generation of virtual machine code at startup. In *Proceedings of the OOPSLA '99 Workshop on Simplicity, Performance, and Portability in Virtual Machine Design*. Sun Microsystems, Inc., November 1999.
- [Gro02] The Open Group. The open group. <http://www.opengroup.org/>, 2002.
- [GV97] C. John Glossner and Stamatis Vassiliadis. The Delft-Java engine. *Lecture Notes in Computer Science*, 1300, 1997.
- [GV99] C. John Glossner and Stamatis Vassiliadis. Delft-Java dynamic translation. In *Proceedings of the 25th Euromicro Conference (EUROMICRO '99)*. Institute of Electrical and Electronics Engineers (IEEE), 1999.
- [Hac99] Scot Hacker. The de facto hardware monopoly. *Byte Magazine*, July 1999.
- [Har97] J. Hart. *Win32 System Programming*. Addison Wesley Longman, 1997.
- [HB89] Colin Hunter and John Banning. DOS at RISC. *BYTE*, 14(12):361–368, 1989.
- [HGH96] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In IEEE, editor, *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, December 2–4, 1996, Paris, France*, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.
- [HH97] Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

- [Hos95] Mat Hostetter. Ardi's dynamically compiling 68lc040 emulator. [http://www.ardi.com/ardi/syn68k\\_paper.html](http://www.ardi.com/ardi/syn68k_paper.html), October 1995.
- [How00] Miles Howson. An Intel Itanium backend for Dynamite. M.Sc., The University of Manchester, October 2000.
- [HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.
- [Int87] Intel. *80386 Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1987.
- [Int01] Intel. *Intel Itanium Architecture Software Developer's Manual*. Intel Corporation, Santa Clara, CA, USA, 2001.
- [KCW98] Edmund J. Kelly, Robert F. Cmelik, and Malcolm John Wing. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. United States Patent 5,832,205, November 1998.
- [Kei01] Gregg Keizer. VMware workstation 3.0. *CNET*, December 2001.
- [Kep96] David Keppel. *Runtime Code Generation*. PhD thesis, University of Washington, 1996.
- [KG97] Andreas Krall and Reinhard Grafl. CACAO — A 64-bit JavaVM Just-in-Time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, November 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.
- [Kla00] Alexander Klaiber. The technology behind crusoe processors. White paper, Transmeta Corporation, January 2000.
- [KMJP00] David Keppel, David MacKenzie, Arne Henrik Juul, and Francois Pinard. *time*. The Free Software Foundation, December 2000.
- [Knu91] Donald E. Knuth. *Artificial Intelligence and Mathematical Theory of Computation*, chapter Textbook examples of recursion, pages 207–229. Academic Press, 1991.

- [Kra98] Andreas Krall. Efficient JavaVM Just-in-Time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, October 1998. IFIP,ACM,IEEE, North-Holland.
- [Law00] Kevin P. Lawton. The Plex86 internals guide. <ftp://plex86.org/docs/Plex86Internals.ps>, 2000.
- [LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-Oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 36–44, 1998.
- [LS96] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, pages 226–237, 1996.
- [LYK<sup>+</sup>99] SeungIl Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soomook Moon, Kemal Ebcioglu, and Erik Altman. On-demand translation of Java exception handlers in the LaTTe JVM Just-in-Time compiler. In *Workshop on Binary Translation*, October 1999.
- [LYK<sup>+</sup>00] SeungIl Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soomook Moon, Kemal Ebcioglu, and Erik R. Altman. Efficient Java exception handling in Just-in-Time compilation. In *Java Grande*, pages 1–8, 2000.
- [Mic01a] Visual J++ home page. <http://msdn.microsoft.com/visualj/>, 2001.
- [Mic01b] Windows NT home. <http://www.microsoft.com/ntserver/>, October 2001.
- [MMBC97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In USENIX, editor, *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, June 16–19, 1997. Portland, Oregon, pages 1–20, Berkeley, CA, USA, June 1997. USENIX.

- [MO98] Harlan McGhan and Mike O'Connor. Computing practices: Pico-Java: A direct execution engine for Java bytecode. *IEEE Computer*, 31(10):22–30, October 1998.
- [MOS<sup>+</sup>98] Satoshi Matsuoka, Hirotaka Ogawa, Kouya Shimura, Yasunori Kimura, Koichito Hotta, and Hiromitsu Takagi. OpenJIT - a reflective Java JIT compiler. In Jean-Charles Fabre and Shigeru Chiba, editors, *Proceedings of Workshop on Reflective Programming in C++ and Java*, October 1998.
- [Nic98] James Niccolai. Symantec aims for enterprise with Java development tool. *JavaWorld: IDG's magazine for the Java community*, 3(8), August 1998.
- [Nor04] Geraint North. Shared code caching method and apparatus for program code conversion. US Patent Application Number 20050015758, March 2004.
- [OAHH03a] Daniel Owen, Jonathan Jay Andrews, Miles Philip Howson, and David Haikney. Generating intermediate representations for program code conversion. GB Patent Number GB2411990, November 2003.
- [OAHH03b] Daniel Owen, Jonathan Jay Andrews, Miles Philip Howson, and David Haikney. Generating intermediate representations for program code conversion. GB Patent Number GB2401217, November 2003.
- [OAHH04] Daniel Owen, Jonathan Jay Andrews, Miles Philip Howson, and David Haiken. Improved architecture for generating intermediate representations for program code conversion. Worldwide Patent Number WO2004097631, April 2004.
- [Oka02] Tetsuya Okado. Dhrystone benchmark in Java. <http://www.c-creators.co.jp/okayan/DhrystoneApplet/>, May 2002.
- [OKN01] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. A study of exception handling and its dynamic optimization in Java. In

- ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, pages 83–95. IBM Japan, Association for Computing Machinery Inc. (ACM), October 2001.
- [Owe00] Daniel Owen. *Optimisation of memory reference patterns in dynamic binary translators*. PhD thesis, The University of Manchester, 2000.
- [PTB<sup>+</sup>97] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A Way Ahead of Time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 41–54, Berkeley, June16–20 1997. Usenix Association.
- [PVC01] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 1–12. Sun Microsystems Inc., The USENIX Association, April 2001.
- [PW97] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompile in Java (does bytecode reveal source?). In USENIX, editor, *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 16–19, 1997. Portland, Oregon*, pages 185–197, Berkeley, CA, USA, June 1997. USENIX.
- [RBDH97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the SimOS machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.
- [Ric00] Adam Richter. *DLOPEN(3)*, Linux programmer’s manual edition, November 2000.
- [Rog99] Ian Rogers. Dynamic compilation of Java bytecodes. M.Phil, The University of Manchester, 1999.
- [RRJ99] Ramesh Radhakrishnan, Juan Rubio, and Lizy Kurian John. Characterization of Java applications at bytecode and Ultra-SPARC machine code levels. In *Proceedings of IEEE International Conference on Computer Design*, pages 281–284, 1999.

- [RRS99] Ian Rogers, Alasdair Rawsthorne, and Jason Souloglou. Exploiting hardware resources: Register assignment across method boundaries. In *Workshop on Hardware Support for Objects and Microarchitecture for Java in conjunction with ICCD'99*, 1999.
- [RSJ<sup>+</sup>03] Alasdair Rawsthorne, Jason Souloglou, John Sandham John, Daniel Owen, and Alex Brown. Block translation optimizations for program code conversation. US Patent Application Number 20040255279, April 2003.
- [SCK<sup>+</sup>92] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Digital Technical Journal of Digital Equipment Corporation*, 4(4):137–152, Fall 1992.
- [SCK<sup>+</sup>93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.
- [SE94] Amitabh Srivastava and Alan Eustace. ATOM - a system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, 1994.
- [SEV01] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, Microsoft Corporation, April 2001.
- [Sha97] Alec Sharp. *Smalltalk by Example*. McGraw-Hill, 1997.
- [SN03] John Sandham and Geraint North. Low overhead flag emulation during code conversion. GB Patent Number GB2388218, February 2003.
- [Sof02] FWB Software. RealPC ©version 1.1.1 for Solaris. [http://www.fwb.com/emu/rpc\\_solaris.htm](http://www.fwb.com/emu/rpc_solaris.htm), 2002.
- [Sol01] Insignia Solutions. Jeode platform white paper. White paper, Insignia Solutions, December 2001.

- [Sou96] Jason Souloglou. A framework dynamic binary translation. M.Phil, The University of Manchester, 1996.
- [SPA92] *The SPARC Architecture Manual: Version 8*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.
- [SPE98] SpecJVM benchmark. <http://www.spec.org/osg/jvm98/>, 1998.
- [SPE00] Spec CPU 95 benchmarks. <http://www.spec.org/osg/cpu95/>, 2000.
- [Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, first edition, 1986.
- [Sun95] Sun Microsystems Inc. *The Java Virtual Machine Specification*, 1.0 beta edition, August 1995.
- [Sun96] Sun Microsystems Inc. *Java Native Interface Specification*, November 1996. Release 1.1.
- [Sun99a] Java 2 SDK, standard edition. <http://java.sun.com/products/jdk/1.2/>, July 1999.
- [Sun99b] Sun Microsystems Inc. *The Java HotSpot Performance Engine Architecture*, April 1999. Sun white paper.
- [Sun01] Sun Microsystems Inc. *The Java Virtual Machine Debug Interface Reference*, 2001.
- [Sun02a] Sun Microsystems Inc. *Java TM 2 Platform, Standard Edition, v 1.3.1 API Specification*, 2002.
- [Sun02b] Sun Microsystems Inc. *Javap — The Java Class File Disassembler*, 2002.
- [Tow98] Tower Technology. TowerJ 2.0 Java bytecode compiler, 1998. The (commercial) TowerJ compiler translates Java source and Java bytecodes to Java bytecodes, native executables, or native shared libraries, for Windows NT/95, SPARC Solaris and SunOS, IBM AIX, HP/UX on PA/RISC, SGI IRIX and Linux.
- [Tra97] Eric Traut. Building the virtual PC. *BYTE*, November 1997.

- [Tra01] Future directions. [http://www.transmeta.com/technology/benchmarks/future\\_direct.html](http://www.transmeta.com/technology/benchmarks/future_direct.html), 2001.
- [Tur96] Jim Turley. Alpha runs x86 code with FX!32. *Microprocessor Report*, 10(3), March 1996.
- [UC00a] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 30–40. ACM Press, January 2000.
- [UC00b] David Ung and Cristina Cifuentes. Optimising hot paths in a dynamic binary translator. In *Workshop on Binary Translation in conjunction with International Conference on Parallel Architectures and Compilation Techniques (PACT) 2000*, October 2000.
- [US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 227–242, New York, NY, 1987. ACM Press.
- [VMw99] Technical white paper. Technical report, VMware Inc., 3145 Porter Drive, Building F, Palo Alto, CA 94303 USA, February 1999.
- [VMw02] VMware workstation 3.0. <http://www.vmware.com/>, 2002.
- [VR01] Frédéric Vivien and Martin C. Rinard. Incrementalized pointer and escape analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2001.
- [Wei84] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM (CACM)*, 27(10):1013–1030, October 1984.
- [WEMW99] Greg Wright, Ahmed El-Mahdy, and Ian Watson. Dynamic Java threads on the Jamaica single-chip multiprocessor. In *Workshop on Hardware Support for Objects and Microarchitecture for Java in conjunction with ICCD 2000*. IEEE, 1999.

- [Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.
- [Wil02] Tim Wilkinson. Kaffe. <http://www.kaffe.org>, 2002.
- [WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.
- [WWEM99] Ian Watson, Greg Wright, and Ahmed El-Mahdy. VLSI Architecture Using Lightweight Threads (VAULT) - choosing the instruction set architecture. In *Workshop on Hardware Support for Objects and Microarchitecture for Java in conjunction with ICCD'99*. IEEE, 1999.
- [YMP<sup>+</sup>99] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik R. Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *IEEE PACT*, pages 128–138, 1999.
- [ZT00] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Computer*, 33(3):47–52, March 2000.